PREVENTING EXTENDING AND OVERRIDING

- Final is used to apply restrictions on class, method, and variable.
- The final class can't be inherited, final method can't be overridden,
- and final variable value can't be changed

PREVENTING EXTENDING AND OVERRIDING

- Standardization: Some classes perform standard functions and they are not meant to be modified e.g. classes performing various functions related to string manipulations or mathematical functions etc.

- Security reasons: Sometimes we write classes which perform various authentication and password related functions and we do not want them to be altered by anyone else.

ABSTRACT CLASS

- An abstract class cannot be used to create objects.

- Some Rules:
  - An abstract class cannot be instantiated using the new operator.
  - An abstract class can contain abstract methods, which are implemented in concrete subclasses.
  - An abstract method cannot be contained in a nonabstract class

INTERFACE

- An interface is a class-like construct that contains only constants and abstract methods

- As with an abstract class, you cannot create an instance from an interface using the new operator

- All data fields are public static final and all methods are public abstract

```
public interface T {
    public static final int K = 1;

    public abstract void p();
}
```

Equivalent

```
public interface T {
    int K = 1;

    void p();
}
```
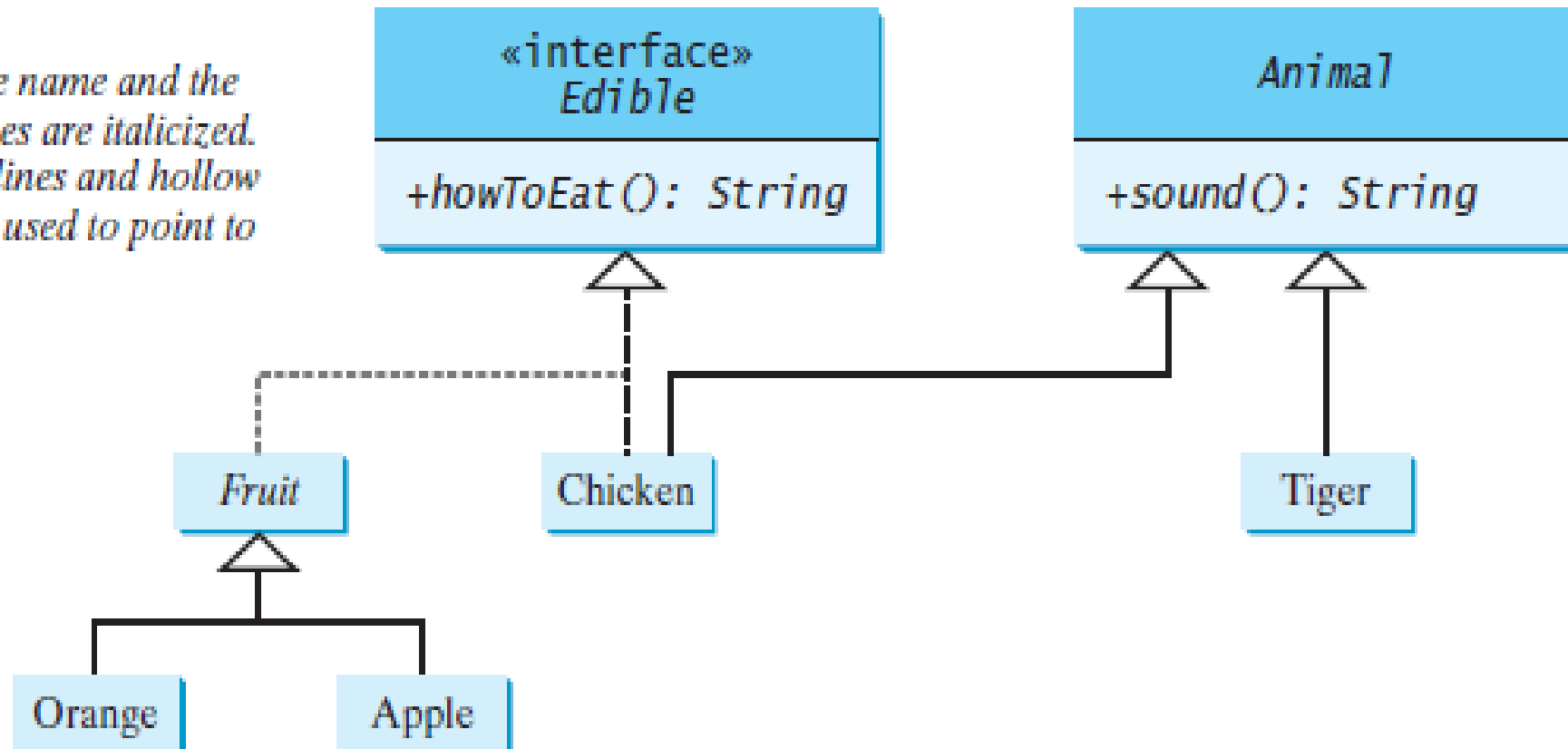
INTERFACE

- It is used to achieve total abstraction.

- Since java does not support multiple inheritance in case of class, but by using interface it can achieve multiple inheritance .

INTERFACE

INTERFACES VS. ABSTRACT CLASSES

- A class can implement multiple interfaces, but it can only extend one superclass.
- An interface can be used more or less the same way as an abstract class

	Variables	Constructors	Methods
Abstract class	No restrictions.	Constructors are invoked by subclasses through constructor chaining. An abstract class cannot be instantiated using the new operator.	No restrictions.
Interface	All variables must be `public static final`.	No constructors. An interface cannot be instantiated using the new operator.	All methods must be public abstract instance methods

EXCEPTION-HANDLING

- Exception handling enables a program to deal with exceptional situations and continue its normal execution.

- In Java, runtime errors are thrown as exceptions. An exception is an object that represents an error or a condition that prevents execution from proceeding normally.

- If the exception is not handled, the program will terminate abnormally.

CATCHING EXCEPTIONS

```
try {
        statements; // Statements that may throw exceptions
}
catch (Exception1 exVar1) {
        // handler for exception1;
}
```

HOW TO THROW EXCEPTIONS IN JAVA

- throw new Exception("Exception message");

TYPES OF EXCEPTIONS

- Checked exception (compile time exception)

  - Checked exceptions must be caught and handled during compile time. If the compiler does not see a try or catch block or throws keyword to handle a checked exception, it throws a compilation error. Checked exceptions are generally caused by faults outside code like missing files, invalid class names, and networking errors.

- Unchecked exception (runtime exception)

  - Unchecked exceptions do not need to be explicitly handled; they occur at the time of execution, also known as run time. These exceptions can usually be avoided by good coding practices. They are typically caused by programming bugs, such as logic errors or improper use of APIs. These exceptions are ignored at the time of compilation

WHAT IS DEBUGGING?

- Broadly, debugging is the process of detecting and correcting errors in a program.

- The debugger is a powerful tool, which lets you find bugs a lot faster by providing an insight into the internal operations of a program. This is possible by pausing the execution and analyzing the state of the program by thorough examination of variables and how they are changed line by line.

DEBUGGING

- Set breakpoints
- Run the program in debug mode
- Step through the program

The Guessing Game

Summary:

The guessing game involves a 'game' object and three 'player' objects. The game generates a random number between 0 and 9, and the three player objects try to guess it. (We didn't say it was a really *exciting* game.)

Classes:

```
GuessGame.class      Player.class      GameLauncher.class
```

The Logic:

1) The GameLauncher class is where the application starts; it has the main() method.

2) In the main() method, a GuessGame object is created, and its startGame() method is called.

3) The GuessGame object's startGame() method is where the entire game plays out. It creates three players, then "thinks" of a random number (the target for the players to guess). It then asks each player to guess, checks the result, and either prints out information about the winning player(s) or asks them to guess again.
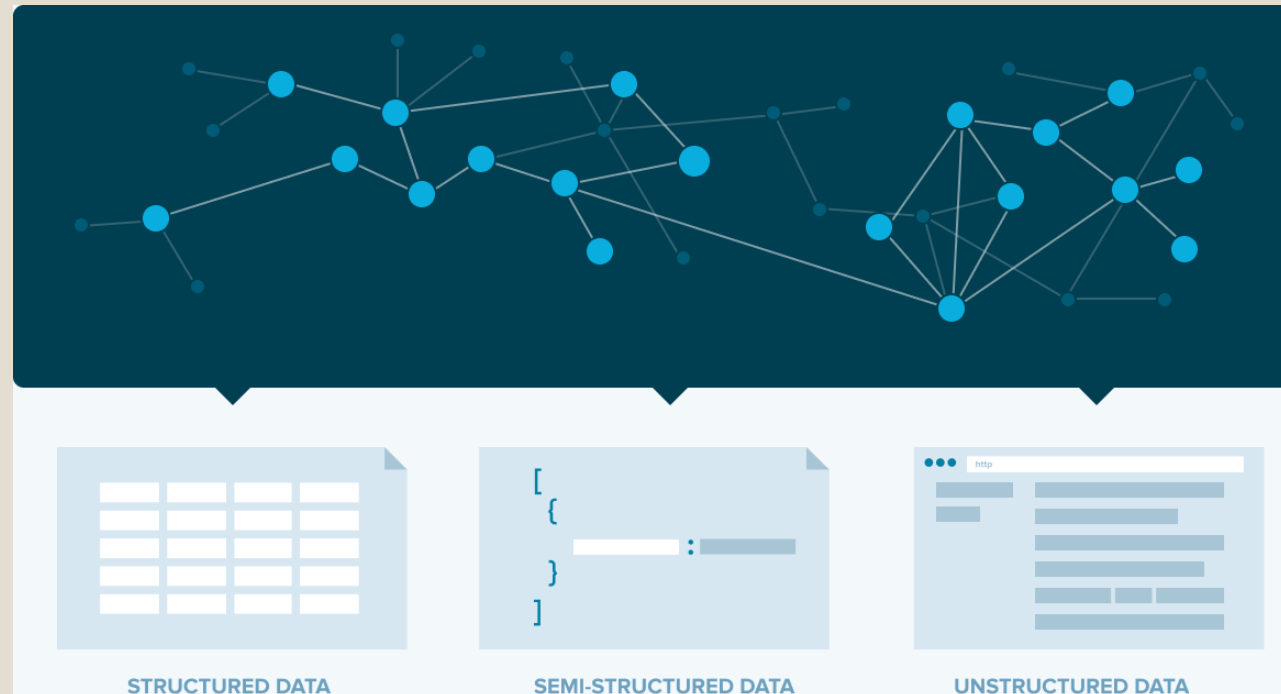
Data Structure
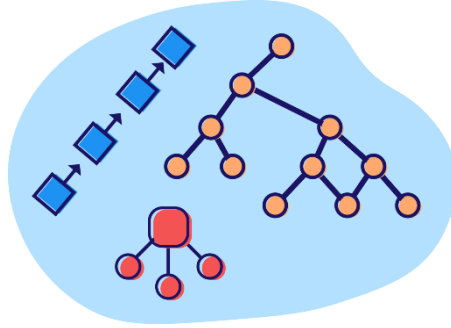
Collections

Any group of individual objects which are represented as a single unit is known as the collection of the objects
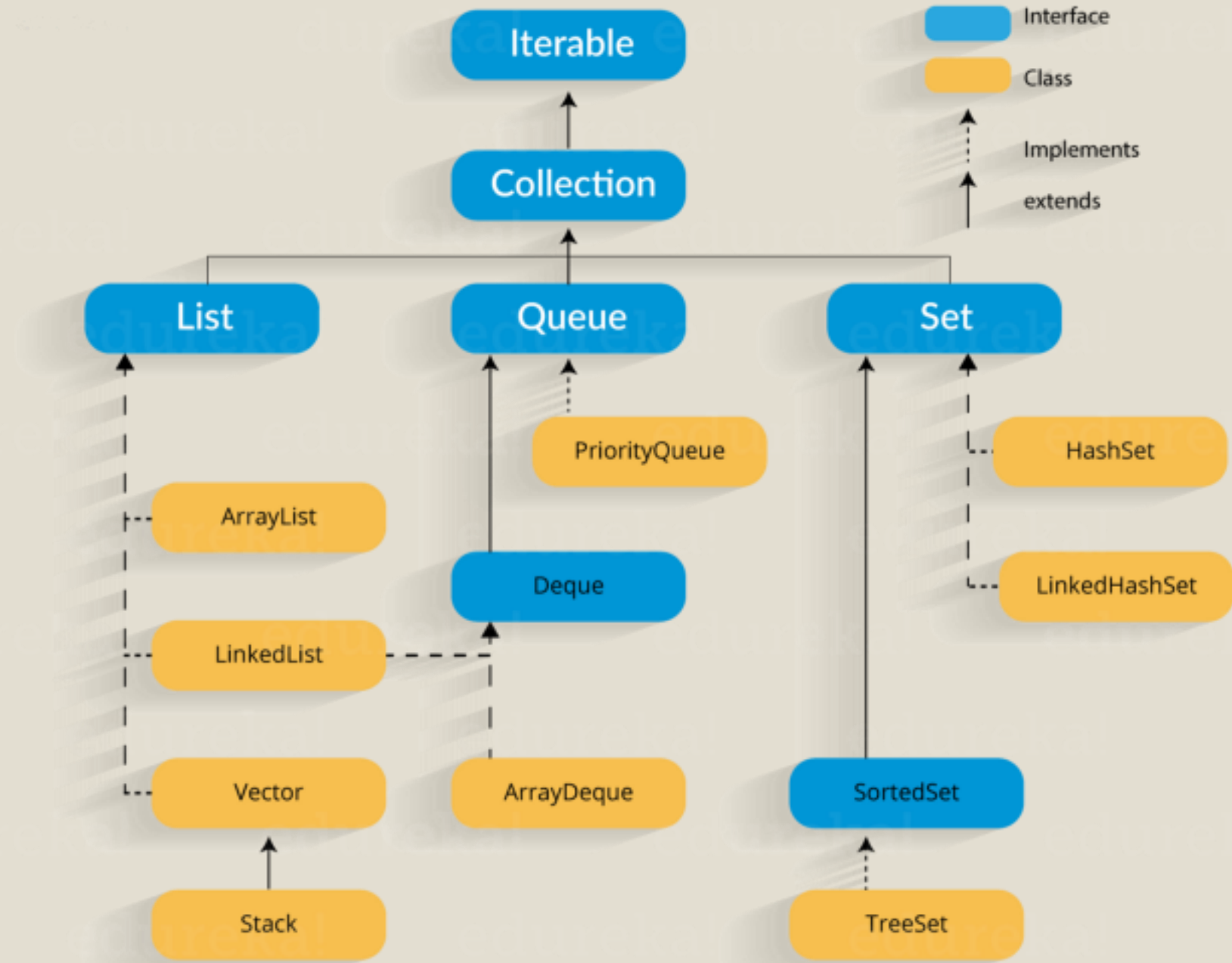
Collections

Java Collections can achieve all the operations we perform on data, such as searching, sorting, insertion, manipulation, and deletion.

STRUCTURED DATA   SEMI-STRUCTURED DATA   UNSTRUCTURED DATA

Advantages of the Collection Framework

- Consistent API: The API has a basic set of interfaces like Collection, Set, List, or Map, all the classes (ArrayList, LinkedList, Vector, etc) that implement these interfaces have some common set of methods.

- Reduces programming effort: A programmer doesn't have to worry about the design of the Collection but rather he can focus on its best use in his program. Therefore, the basic concept of Object-oriented programming (i.e.) abstraction has been successfully implemented.

Array vs Collection

- Arrays are simple constructs with linear storage of fixed size. Therefore they can only store a given number of elements.
- Arrays can hold only the same type of data (homogeneous).

- Collections are more sophisticated and flexible.
- They are resizable: we can add any number of elements to a collection.
- They can hold both homogeneous and heterogeneous elements.

Array vs Collection

- Arrays are fixed size
- ArrayList's size auotomatically adjusted
- Arrays can hold primitives and object
- ArrayList can hold only objects
- Arrays can be multi dimensional
- ArrayList cannot be multi-dimentional
- Array is a build in data structure
- ArrayList is implementing class of List interface in Collection framework

List Interface

- List interface is the child interface of the Collection interface.

- The list is an ordered collection

- They may contain duplicate elements.

- Elements can be inserted or accessed by their position in the list using a zero-based index.

# List Interface

- List interface is implemented by the classes

  - ArrayList

  - LinkedList

  - Vector

  - Stack.

JAVA ARRAYLIST

- The ArrayList class is a resizable array, which can be found in the java.util package.

- The difference between a built-in array and an ArrayList in Java, is that the size of an array cannot be modified (if you want to add or remove elements to/from an array, you have to create a new one). While elements can be added and removed from an ArrayList whenever you want.

- ArrayList<String> cars = new ArrayList<String>(); // Create an ArrayList object

Java ArrayList

- In Java, we need to declare the size of an array before we can use it. Once the size of an array is declared, it's hard to change it.

- To handle this issue, we can use the ArrayList class. It allows us to create resizable arrays.

- Unlike arrays, arraylists can automatically adjust its capacity when we add or remove elements from it. Hence, arraylists are also known as dynamic arrays.
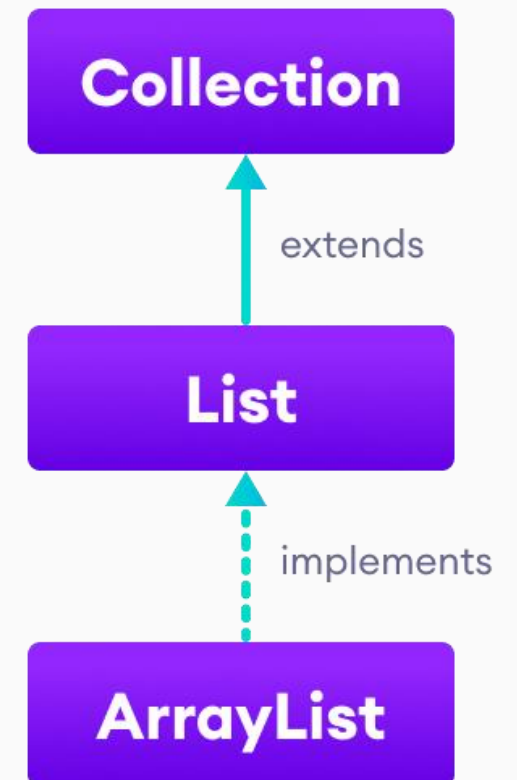
Basic Operations on ArrayList

- ArrayList<String> languages = new ArrayList<>();

- Add Elements to an ArrayList

  - languages.add("Java"); // add() method without the index parameter

  - languages.add(1, "C++"); // insert element at position 1

- Access ArrayList Elements .get(1);
- languages.set(2, "JavaScript");
- .remove(2); OR languages.remove("Java");

Basic Operations on ArrayList

- Convert an array to an list -> Arrays.asList(arr)

Linked List

- Singly linked list

  - Singly linked lists contain nodes which have a data field as well as 'next' field, which points to the next node in line of nodes.

  - Operations that can be performed on singly linked lists include insertion, deletion and traversal.
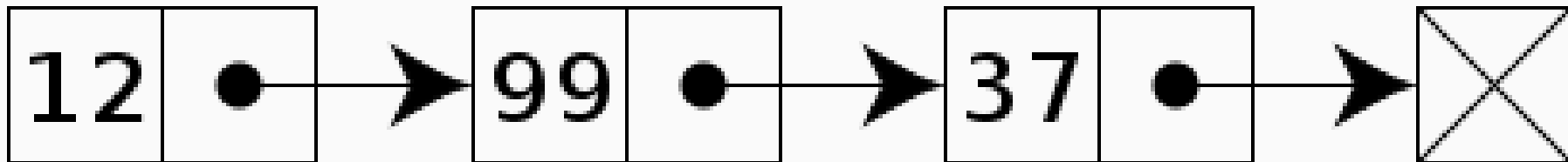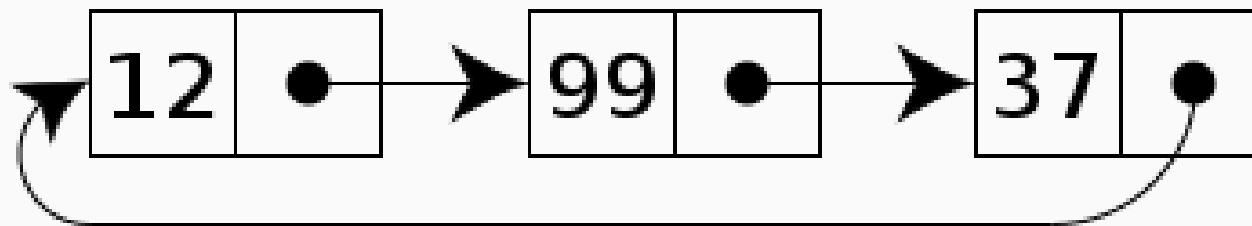
Linked List

- Doubly linked list

    - In a 'doubly linked list', each node contains, besides the next-node link, a second link field pointing to the 'previous' node in the sequence.

    - The two links may be called 'forwards' and 'backwards', or 'next' and 'prev'('previous').
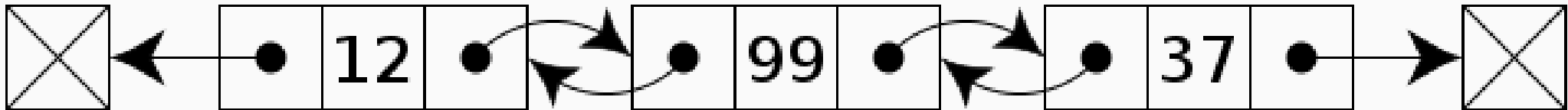
Linked List

- Circular linked list

  - In the last node of a list, the link field often contains a null reference, a special value is used to indicate the lack of further nodes.

  - A less common convention is to make it point to the first node of the list; in that case, the list is said to be 'circular' or 'circularly linked'; otherwise, it is said to be 'open' or 'linear'

  - It is a list where the last pointer points to the first node.

Linked List Examples

- Previous and next page (URL) in a web browser

- Music Player

- Escalator

- A telephone chain

LINKED LIST

- ADVANTAGE: Dynamicity and ease of insertions and deletions.

- DISADVANTAGE: Nodes cannot be accessed directly instead we need to start from the head and follow through the link to reach a node we wish to access

LINKED LIST

Methods	Descriptions
addFirst()	adds the specified element at the beginning of the linked list
addLast()	adds the specified element at the end of the linked list
getFirst()	returns the first element
getLast()	returns the last element
removeFirst()	removes the first element
removeLast()	removes the last element
peek()	returns the first element (head) of the linked list
poll()	returns and removes the first element from the linked list

LinkedList Vs. ArrayList

LinkedList

Implements List, Queue, and Deque interfaces.

Stores 3 values (**previous address**, **data,** and **next address**) in a single position.

Provides the doubly-linked list implementation.

Whenever an element is added, prev and next address are changed.

To access an element, we need to iterate from the beginning to the element.

ArrayList

Implements List interface.

Stores a single value in a single position.

Provides a resizable array implementation.

Whenever an element is added, all elements after that position are shifted.

Can randomly access elements using indexes.

How are linked lists more efficient than arrays? (1/3)

- **Insertion and Deletion**

  - Insertion and deletion process is expensive in an array as the room has to be created for the new elements and existing elements must be shifted.

  - But in a linked list, the same operation is an easier process, as we only update the address present in the next pointer of a node.

How are linked lists more efficient than arrays? (2/3)

- **Dynamic Data Structure**

  - Linked list is a dynamic data structure that means there is no need to give an initial size at the time of creation as it can grow and shrink at runtime by allocating and deallocating memory.

  - Whereas, the size of an array is limited as the number of items is statically stored in the main memory.

How are linked lists more efficient than arrays? (3/3)

- **No wastage of memory**

    - As the size of a linked list can grow or shrink based on the needs of the program, there is no memory wasted because it is allocated in runtime.

    - In arrays, if we declare an array of size 10 and store only 3 elements in it, then the space for 3 elements is wasted. Hence, chances of memory wastage is more in arrays.

We can create a LinkedList using interfaces in Java

// create linkedlist using List

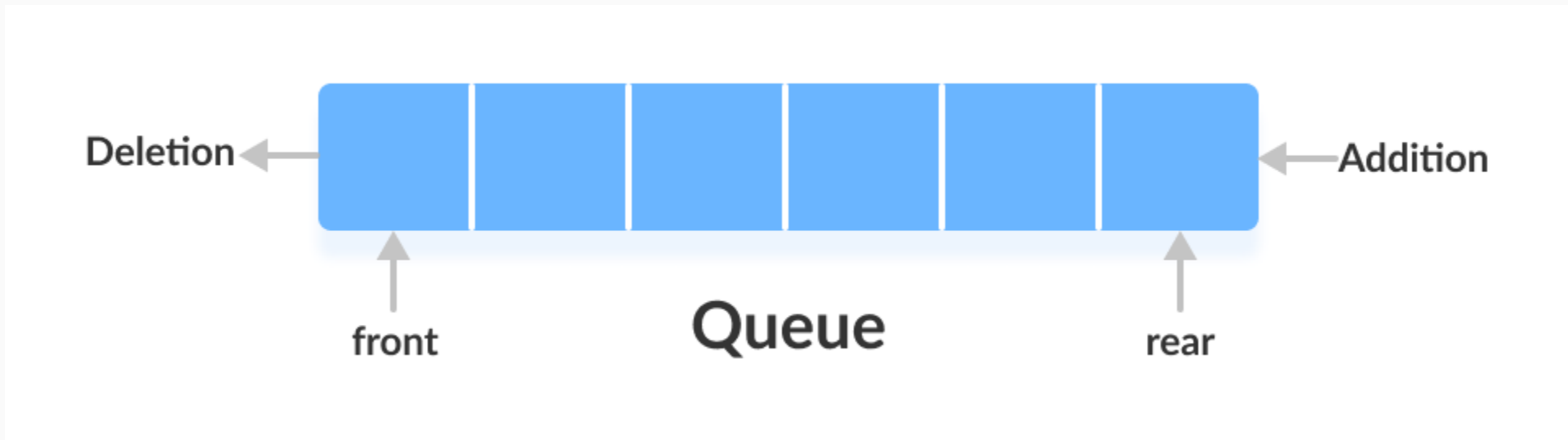- List<String> animals1 = new LinkedList<>();

// creating linkedlist using Queue

- Queue<String> animals2 = new LinkedList<>();

// creating linkedlist using Deque

- Deque<String> animals3 = new LinkedList<>();

Queue Data Structure

- In queues, elements are stored and accessed in **First In, First Out** manner. That is, elements are **added from the behind** and **removed from the front**.

Methods of the Queue interface

- add() - Inserts the specified element into the queue. If the task is successful, add() returns true, if not it throws an exception.
- offer() - Inserts the specified element into the queue. If the task is successful, offer() returns true, if not it returns false.
- element() - Returns the head of the queue. Throws an exception if the queue is empty.
- peek() - Returns the head of the queue. Returns null if the queue is empty.
- remove() - Returns and removes the head of the queue. Throws an exception if the queue is empty.
- poll() - Returns and removes the head of the queue. Returns null if the queue is empty.

Queue interface exmple

```java
// Creating Queue using the LinkedList class
Queue<Integer> numbers = new LinkedList<>();

// offer elements to the Queue
numbers.offer(1);
numbers.offer(2);
System.out.println("Queue: " + numbers);

// Access elements of the Queue
int accessedNumber = numbers.peek();
System.out.println("Accessed Element: " + accessedNumber);
```

PriorityQueue

- Unlike normal queues, priority queue elements are retrieved in sorted order.

- Suppose, we want to retrieve elements in the ascending order. In this case, the head of the priority queue will be the smallest element. Once this element is retrieved, the next smallest element will be the head of the queue.

- It is important to note that the elements of a priority queue may not be sorted. However, elements are always retrieved in sorted order.

Collection

extends

**Queue**

implements

PriorityQueue
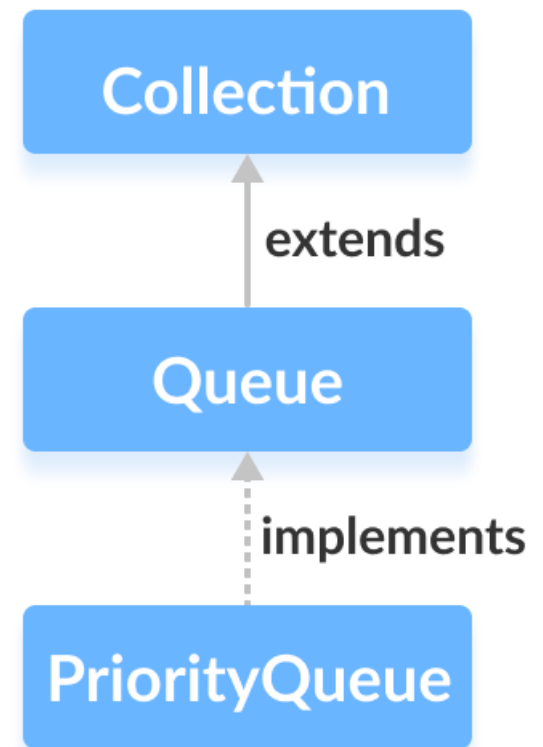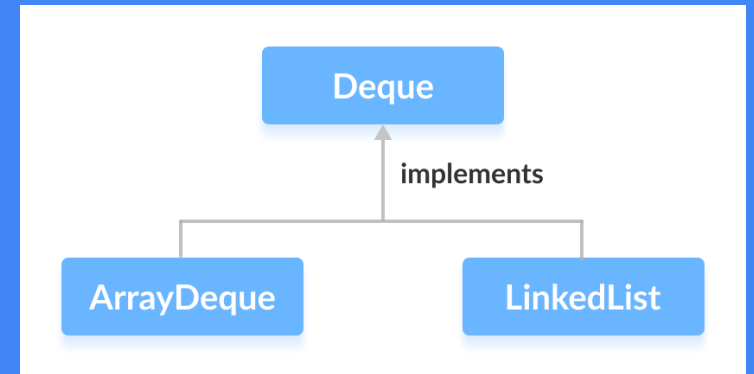
Deque

- In a regular queue, elements are added from the rear and removed from the front. However, in a deque, we can insert and remove elements from both front and rear.
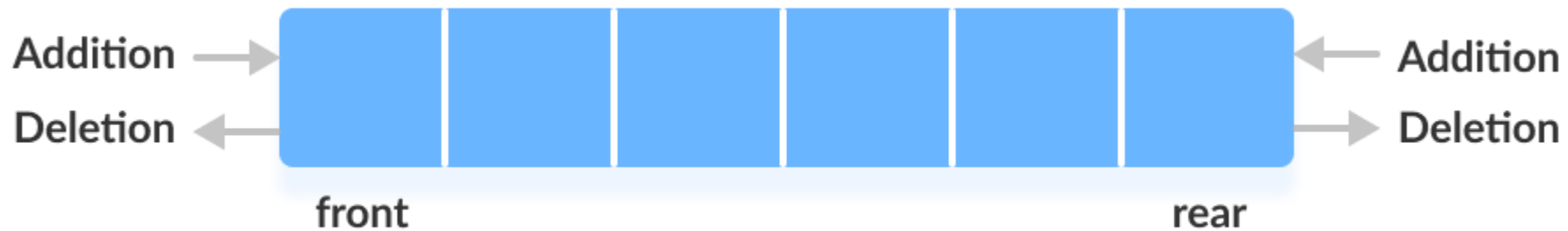
Deque

- offerFirst() - Adds the specified element at the beginning of the deque.
- offerLast() - Adds the specified element at the end of the deque.

- peekFirst() - Returns the first element of the deque. Returns null if the deque is empty.
- peekLast() - Returns the last element of the deque. Returns null if the deque is empty.

- pollFirst() - Returns and removes the first element of the deque. Returns null if the deque is empty.
- pollLast() - Returns and removes the last element of the deque. Returns null if the deque is empty.
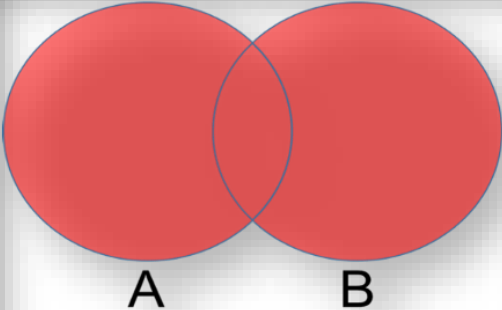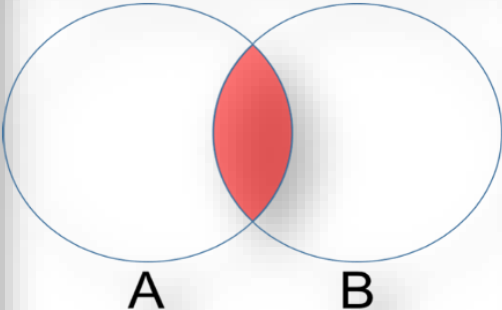
Set Interface
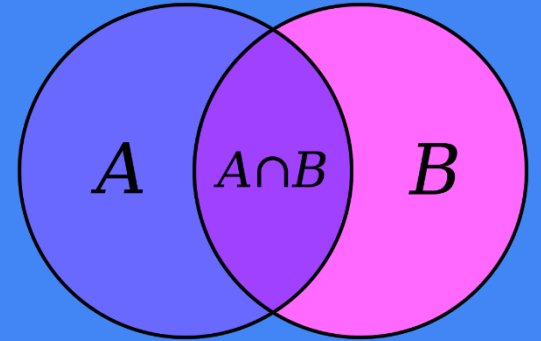
- A Set is a collection that cannot contain duplicate elements
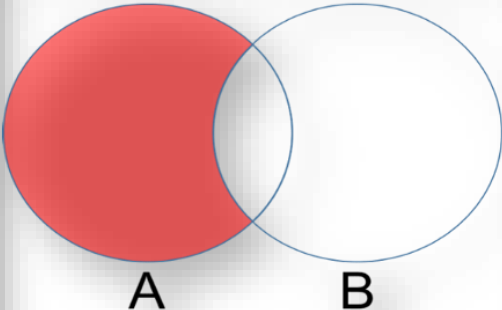- There are three main implementations of the Set interface: HashSet, TreeSet, and LinkedHashSet
- HashSet, which stores its elements in a hash table, is the best-performing implementation.
- LinkedHashSet extends the HashSet class and implements the Set interface. Like HashSet, It also contains unique elements. It maintains the insertion order and permits null elements.

	HashSet	LinkedHashSet	TreeSet
How they work internally?	HashSet uses HashMap internally to store it's elements.	LinkedHashSet uses LinkedHashMap internally to store it's elements.	TreeSet uses TreeMap internally to store it's elements.
Order Of Elements	HashSet doesn't maintain any order of elements.	LinkedHashSet maintains insertion order of elements. i.e elements are placed as they are inserted.	TreeSet orders the elements according to supplied Comparator. If no comparator is supplied, elements will be placed in their natural ascending order.
Performance	HashSet gives better performance than the LinkedHashSet and TreeSet.	The performance of LinkedHashSet is between HashSet and TreeSet. It's performance is almost similar to HashSet. But slightly in the slower side as it also maintains LinkedList internally to maintain the insertion order of elements.	TreeSet gives less performance than the HashSet and LinkedHashSet as it has to sort the elements after each insertion and removal operations.
Insertion, Removal And Retrieval Operations	HashSet gives performance of order O(1) for insertion, removal and retrieval operations.	LinkedHashSet also gives performance of order O(1) for insertion, removal and retrieval operations.	TreeSet gives performance of order O(log(n)) for insertion, removal and retrieval operations.

|			
How they compare the elements?	HashSet uses equals() and hashCode() methods to compare the elements and thus removing the possible duplicate elements.	LinkedHashSet also uses equals() and hashCode() methods to compare the elements.	TreeSet uses compare() or compareTo() methods to compare the elements and thus removing the possible duplicate elements.
Null elements	HashSet allows maximum one null element.	LinkedHashSet also allows maximum one null element.	TreeSet doesn't allow even a single null element. If you try to insert null element into TreeSet, it throws NullPointerException.
Memory Occupation	HashSet requires less memory than LinkedHashSet and TreeSet as it uses only HashMap internally to store its elements.	LinkedHashSet requires more memory than HashSet as it also maintains LinkedList along with HashMap to store its elements.	TreeSet also requires more memory than HashSet as it also maintains Comparator to sort the elements along with the TreeMap.
When To Use?	Use HashSet if you don't want to maintain any order of elements.	Use LinkedHashSet if you want to maintain insertion order of elements.	Use TreeSet if you want to sort the elements according to some Comparator.

Set Methods

- add() - adds the specified element to the set
- addAll() - adds all the elements of the specified collection to the set
- remove() - removes the specified element from the set
- removeAll() - removes all the elements from the set that is present in another specified set
- retainAll() - retains all the elements in the set that are also present in another specified set
- clear() - removes all the elements from the set
- size() - returns the length (number of elements) of the set
- toArray() - returns an array containing all the elements of the set
- contains() - returns true if the set contains the specified element
- containsAll() - returns true if the set contains all the elements of the specified collection

Set Operation	Venn Diagram	Interpretation
Union		$A \cup B$, is the set of all values that are a member of A, or B, or both.
Intersection		$A \cap B$, is the set of all values that are members of both A and B.
Difference		$A \setminus B$, is the set of all values of A that are not members of B

Set Operations

- To get the **union** of two sets x and y, we can use x.addAll(y)

- To get the **intersection** of two sets x and y, we can use x.retainAll(y)

- To check if x is a **subset** of v. we can use v.containsAll(x)

A ∩ B = {1, 3}
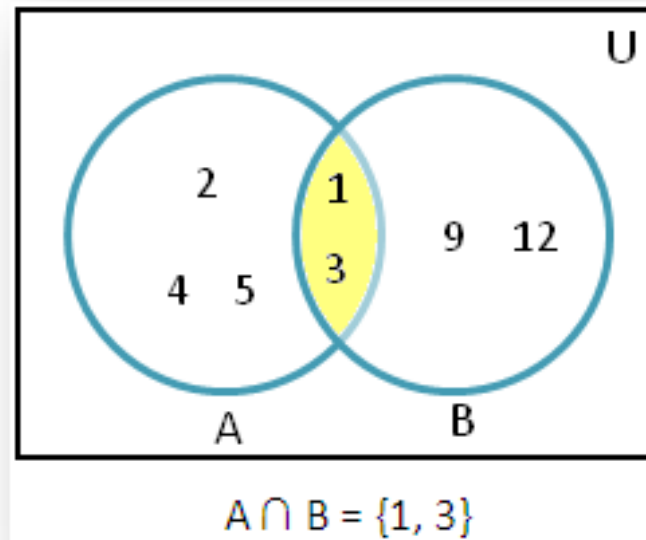
MAP

- A map contains values on the basis of key, i.e. key and value pair.
- Each key and value pair is known as an entry.
- A Map contains unique keys.
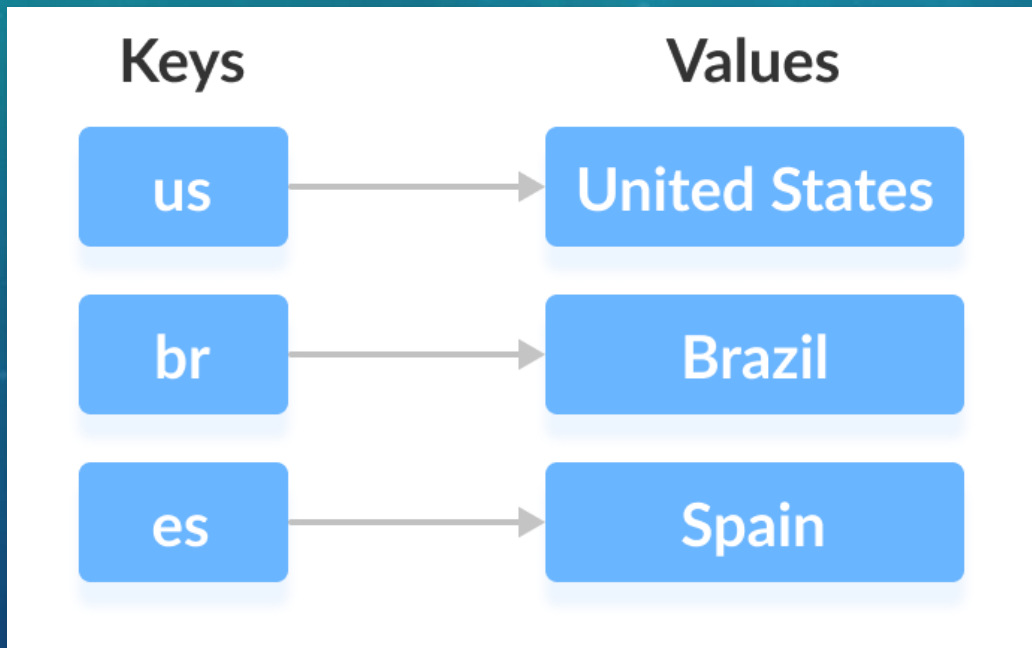
MAP

- Phonebook (Name - Number)
- Dictionary Application (Word - Meaning)
- City - zip code
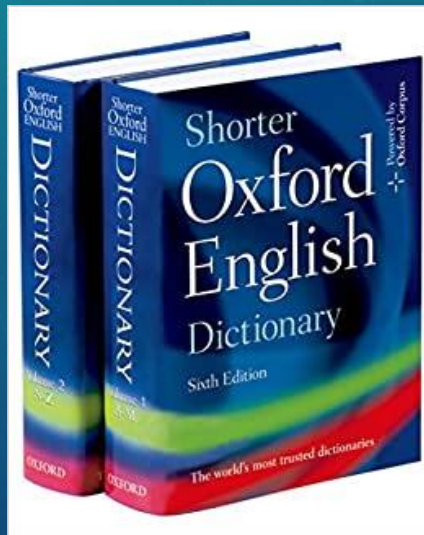
Class	Description
HashMap	It doesn't maintain any order.
LinkedHashMap	It inherits HashMap class. It maintains insertion order.
TreeMap	TreeMap is the implementation of Map and SortedMap. It maintains ascending order.

MAP

- put(Object key, Object value)
    - It is used to insert an entry in the map.

- putIfAbsent(K key, V value)
    - It inserts the specified value with the specified key in the map only if it is not already specified.
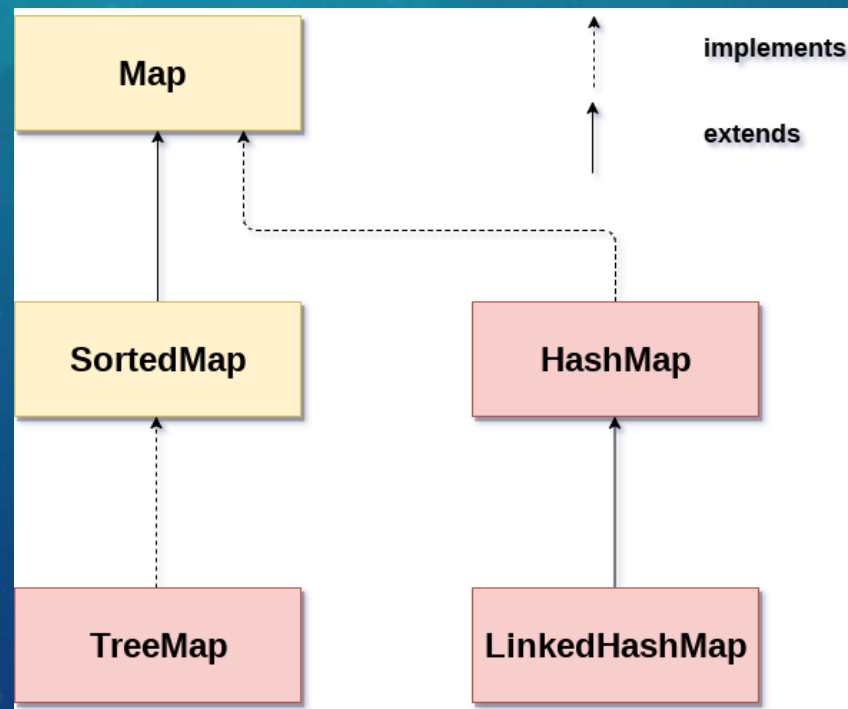
- remove(Object key)
    - It is used to delete an entry for the specified key.

- get(Object key)
    - This method returns the object that contains the value associated with the key.

- replace(K key, V value)
    - It replaces the specified value for a specified key.

MAP

- keySet()
  - It returns the Set view containing all the keys.

- containsValue(Object value)
  - This method returns true if some value equal to the value exists within the map, else return false.

- containsKey(Object key)
  - This method returns true if some key equal to the key exists within the map, else return false.

- size()
  - This method returns the number of entries in the map.

WHEN DO YOU USE SET, LIST, AND MAP IN JAVA?

- Use set when you don't need duplicates,
- Use List when you need order with duplicates,
- Use Map when you need to store key-value pair.

- If you need to access elements frequently using the index:

- If you want to store elements and want them to maintain an order on which they are inserted into a collection:

- If you want to create a collection of unique elements and don't want any duplicates:

- If you need to store data in form of key and value:\

- If you need to access elements frequently using the index use Lists.
  - ArrayList provides faster access if you know the index.

- If you want to store elements and want them to maintain an order on which they are inserted into a collection then go for List again.

- If you want to create a collection of unique elements and don't want any duplicates then choose any Set implementation

- If you need to store data in form of key and value then use the Map interface.

CHALLENGES

- Create an ArrayList with some numbers

  - Try to remove the odds

- Create an 2D ArrayList

  - Implement the add / get / set methods

- Simulate a game where in each round the player with the lowest points is out of the game

PROGRAMMING PATTERNS WITH ARRAYLISTS

- A programming pattern is a general, reusable solution to an often occurring problem.

- While working on the exercises with ArrayList you may have already noticed some repeating patterns.

PROGRAMMING PATTERNS WITH ARRAYLISTS

- Take a list, change all elements in it to a new value. (Map)

  - You have a list of String values and should return a list with all that String values in lowercase (or in uppercase)

  - You have a list of int values and should return a list which each value multiplied by 2

  - You have a list of String values and should return a list of the length of each of those String values

PROGRAMMING PATTERNS WITH ARRAYLISTS - MAP

```java
public static ArrayList<String> mapValues(ArrayList<String> oldValues) {
    ArrayList<String> newValues = new ArrayList<String>();
    for(String value : oldValues) {
        String newValue = // here you would do your operation based on the
oldValue, e.g. value.toUpperCase()
        newValues.add(newValue):
    }

    return newValues;
}
```

PROGRAMMING PATTERNS WITH ARRAYLISTS - MAP

```
public static void mapValues(ArrayList<String> values) {

    for(int i = 0; i < values.size(); i++) {
        String newValue = // here you would do your operation based on the oldValue, e.g. value.toUpperCase()

        newValues.set(i, newValue);
    }
}
```

PROGRAMMING PATTERNS WITH ARRAYLISTS - FILTER

- Take a list, remove all elements that do not fulfill a certain criteria. (Filter)

    - you have a list of String values and should return a list with all the String values of a certain length

    - you have a list of int values and should return a list which only contains the odd numbers

    - you have a list of String values and should return a list of these values without any duplicates

PROGRAMMING PATTERNS WITH ARRAYLISTS - FILTER

```
public static ArrayList<String> filterValues(ArrayList<String> oldValues) {
    ArrayList<String> newValues = new ArrayList<String>();
    for(String value : oldValues) {
        // here we define our condition based on value, e.g. value.length() > 10
        if( … ) {
            newValues.add(value):
        }
    }

    return newValues;
}
```

PROGRAMMING PATTERNS WITH ARRAYLISTS - FILTER

- If the condition depends also on the position of the element in the list, we should use the standard for loop

```
public static ArrayList<String> filterValues(ArrayList<String> oldValues) {
    ArrayList<String> newValues = new ArrayList<String>();
    for(int i = 0; i < oldValues.size(); i++) {
        // here we define our condition based on value and/or index i,
        // e.g. value.length() > 10 or i % 2 == 0 to keep only each second element
        if( ... ) {
            newValues.add(oldValues.get(i)):
        }
    }
    return newValues;
}
```

PROGRAMMING PATTERNS WITH ARRAYLISTS - REDUCE

- Take a list, return one value (Reduce)

  - you have a list of String values and should return one String which is all the values concatenated

  - you have a list of int values and should return their sum

  - you have a list of int values and should return their maximum or minimum

  - you have a list of String and should return the maximum length

PROGRAMMING PATTERNS WITH ARRAYLISTS - REDUCE

```java
public static int reduceValues(ArrayList<Integer> values) {
    int result = 0;
    for(Integer value : values) {
        result = ... // here you add your logic to compute the new result using the current element, e.g. for a sum you would do result = result + value
    }

    return result;
}
```

PROGRAMMING PATTERNS WITH ARRAYLISTS - REDUCE

- If the result depends also on the position of the element in the list, we should use the standard for loop

```
public static int reduceValues(ArrayList<Integer> values) {
    int result = 0;
    for(int i = 0; i < oldValues.size(); i++) {
        int value = oldValues.get(i);
        // here we sum, but before each value is multiplied with its index
        value = value * i;
        result = result + value;
    }
    return result;
}
```

ELEVATOR

- Number of passengers (Max 8)
- 20 Floors (0-19)