JaVA Building Block Stairs (JaBS)
(Java Key Feature: Abstraction)

+ ++ +

+ **"JaBS" is the compilation of Knowledge of Java. Java evolved from the features inherited from C and C++ and polished their features to improve the current demand of programming. This document is solely the property of EWU, CSE. Prepared by Dr. Hasan Mahmood Aminul Islam, Faculty, CSE, EWU.** +

+ ++ +

Abstraction meaning in general: The process of working with ideas rather than their implementation.

Abstraction in Java is one of the most important key features of Java which hides its internal and other implementation details and cleanly separates its interface from the implementation of the other modules. In other words, the user will have just the knowledge of what an entity is doing instead of its internal working.

How to Achieve Abstraction in Java?

In Java, we can achieve Data Abstraction using Abstract classes and interfaces. Interfaces allow 100% abstraction (complete abstraction). Interfaces allow you to abstract the implementation completely.
Abstract classes allow 0 to 100% abstraction (partial to complete abstraction) because abstract classes can contain concrete methods that have the implementation which results in a partial abstraction.

Abstract Classes in Java

- An **Abstract class** is a class whose objects can't be created. An **Abstract** class is created through the use of the **abstract keyword**. It is used to represent a concept.
- An abstract class can have abstract methods (**methods** without body) as well as non-abstract methods or concrete methods (methods with the body). A non-abstract class cannot have abstract methods.
- The class has to be declared as **abstract** if it contains at least one abstract method.
- **An abstract class** does not allow you to create objects of its type. In this case, we can only use the objects of its subclass.
- Using an **abstract class**, we can achieve 0 to 100% abstraction.
- There is always a **default constructor** in an abstract class, it can also have a parameterized constructor.
- The **abstract class** can also contain final and static methods.

Syntax of declaring an abstract class:

```
abstract class ClassName{
    //class body
}
```

JaVA Building Block Stairs (JaBS)
(Java Key Feature: Abstraction)

Abstract Methods in Java

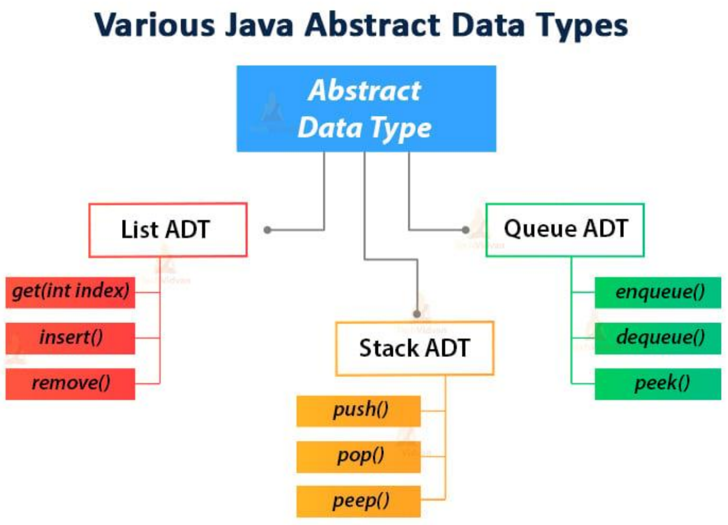
- **Abstract methods** are methods with **no implementation and without a method body.**
- An **abstract** method is declared with an **abstract** keyword.
- The **child classes** which **inherit** the **abstract class** must provide the implementation of these inherited **abstract methods.**

1. List ADT

The List Abstract Data Type is a type of list that contains similar elements in sequential order. The list ADT is a collection of elements that have a linear relationship with each other. A linear relationship means that each element of the list has a unique successor.

The List ADT is an interface, that is, other classes give the actual implementation of the data type. For example, Array Data Structure internally implements the ArrayList class while List Data Structure internally implements the LinkedList class.

List of Java Abstract Data Type



The List *interface of the Java* library specifies 25 different operations/methods. Following are some of the operations that we can perform on the list:

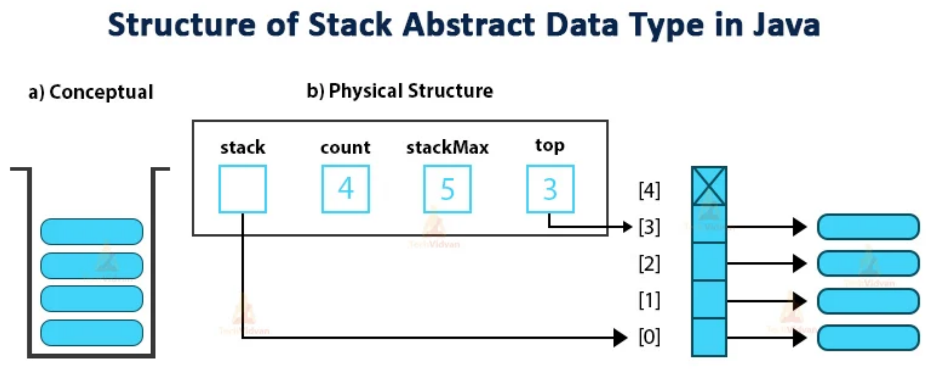
- ❖ **`get(int index):`** Returns an element at the specified index from the list.
- ❖ **`insert():`** Inserts an element at any position.
- ❖ **`remove():`** Removes the first occurrence of any element from a list.
- ❖ **`removeAt():`** Removes the element at a predefined area from a non-empty list.
- ❖ **`Replace():`** Replaces an element by another element.
- ❖ **`size():`** Returns the number of elements of the list.
- ❖ **`isEmpty():`** Returns true if the list is empty, else returns false.
- ❖ **`isFull():`** Returns true if the list is full, else returns false.

2. Stack ADT

A stack is a LIFO ("Last In, First Out") data structure that contains similar elements arranged in an ordered sequence. All the operations in the stack take place at the top of the stack.

- ❖ Stack ADT is a collection of homogeneous data items (elements), in which all insertions and deletions occur at one end, called the top of the stack.
- ❖ In Stack ADT Implementation, there is a pointer to the data, instead of storing the data at each node.
- ❖ The program allocates the memory for the data and passes the address to the stack ADT.
- ❖ The start node and the data nodes encapsulate together in the ADT. Only the pointer to the stack is visible to the calling function.
- ❖ The stack head structure also contains a pointer to the top of the stack and also the count of the number of entries currently in the stack.

The below diagram shows the whole structure of the Stack ADT:



We can perform the following operations on the stack –

- **`push()`:** It inserts an element at the top of the stack if the stack is not full.
- **`pop()`:** It removes or pops an element from the top of the stack if the stack is not empty.
- **`peep()`:** Returns the top element of the stack without removing it.
- **`size()`:** Returns the size of the stack.
- **`isEmpty()`:** If the stack is empty it returns true, else it returns false.
- **`isFull()`:** If the stack is full it returns true, else it returns false.

3. Queue ADT

A Queue is a FIFO ("First In, First Out") data structure that contains similar types of elements arranged sequentially. We can perform the operations on a queue at both ends; insertion takes place at rear end, deletion takes place at the front end.

Queue ADT is a collection in which the arrangement of the elements of the same type is in a sequential manner.

- The design of the Queue abstract data type (ADT) is the same as the basic design of the Stack ADT.
- Each node of the queue contains a void pointer to the data and a link pointer to the next element of the queue. The program allocates the memory for storing the data.

Operations performed on the queue are as follows:

- **`enqueue()`:** It inserts or adds an element at the end of the queue.
- **`dequeue()`:** Removes an element from the front side of the queue.
- **`peek()`:** Returns the starting element of the queue without removing it.
- **`size()`:** This function returns the number of elements in the queue.
- **`isEmpty()`:** If the queue is empty, it returns true, otherwise it returns false.
- **`isFull()`:** If the queue is full, it returns true, otherwise it returns false.

Designing an Abstract Data Type in Java

To design an abstract data type we have to choose good operations and determine how they should behave. Here are a few rules for designing an ADT.

- It's better to combine simple and few operations in powerful ways, rather than a lot of complex operations.

- Each operation in an Abstract Data Type should have a clear purpose and should have a logical behavior rather than a range of special cases. All the special cases would make operation difficult to understand and use.
- The set of operations should be adequate so that there are enough kinds of computations that users likely want to do.
- The type may be either generic, for example, a graph, a list or a set, or it may be domain-specific for example, an employee database, a street map, a phone book, etc. But there should not be a combination of generic and domain-specific features.

Link: (https://docs.oracle.com/javase/8/docs/api/java/util/List.html)

Which Java Abstract Data Type to choose?

Now after having brief knowledge of Java Abstract Data Types, we will discuss the scenarios to choose between either of List, Stack or Queue ADT.

List ADT is a collection of elements and stores them sequentially and which we can access using their indices. We can opt for this ADT in cases that involve indexed or sequential access or removal of elements.

For example, we can use various implementations of List ADT to store data of a list of employees in sorted order for sequential access or removal.

A Stack is a Last In First out data structure, and thus we can use implementations of Stack ADT in scenarios where we need to firstly access the most recently inserted elements.

For example, the function call stack of every programming language has the common requirement of this kind of LIFO data structure where there is a need to execute the most recent function in the stack.

The queue is a First In First Out data structure and we can choose the Queue ADT in scenarios where we need to access the elements in their order of insertion.

For example, one such scenario is request handling by web servers. Web servers make it possible to ensure the fairness of request handling according to their order of arrival by maintaining an internal queue for the requests.