Abstract Data Types

Presentation by **Haya R**

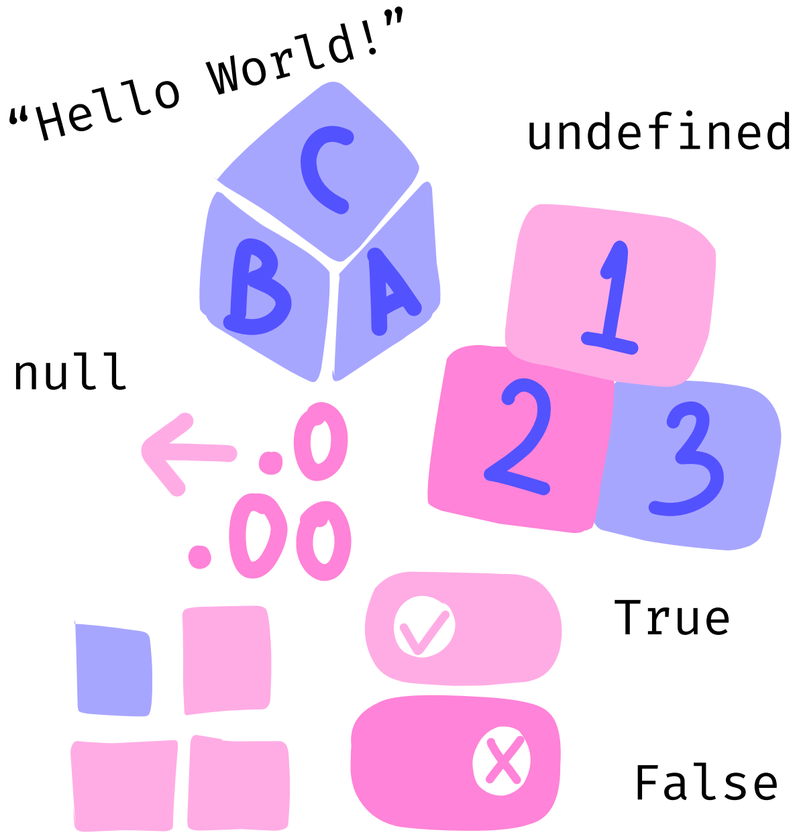
Overview

Preliminary Concepts

- Primitive Data Types
- User Defined Types
- Abstraction
- Interface vs Implementation

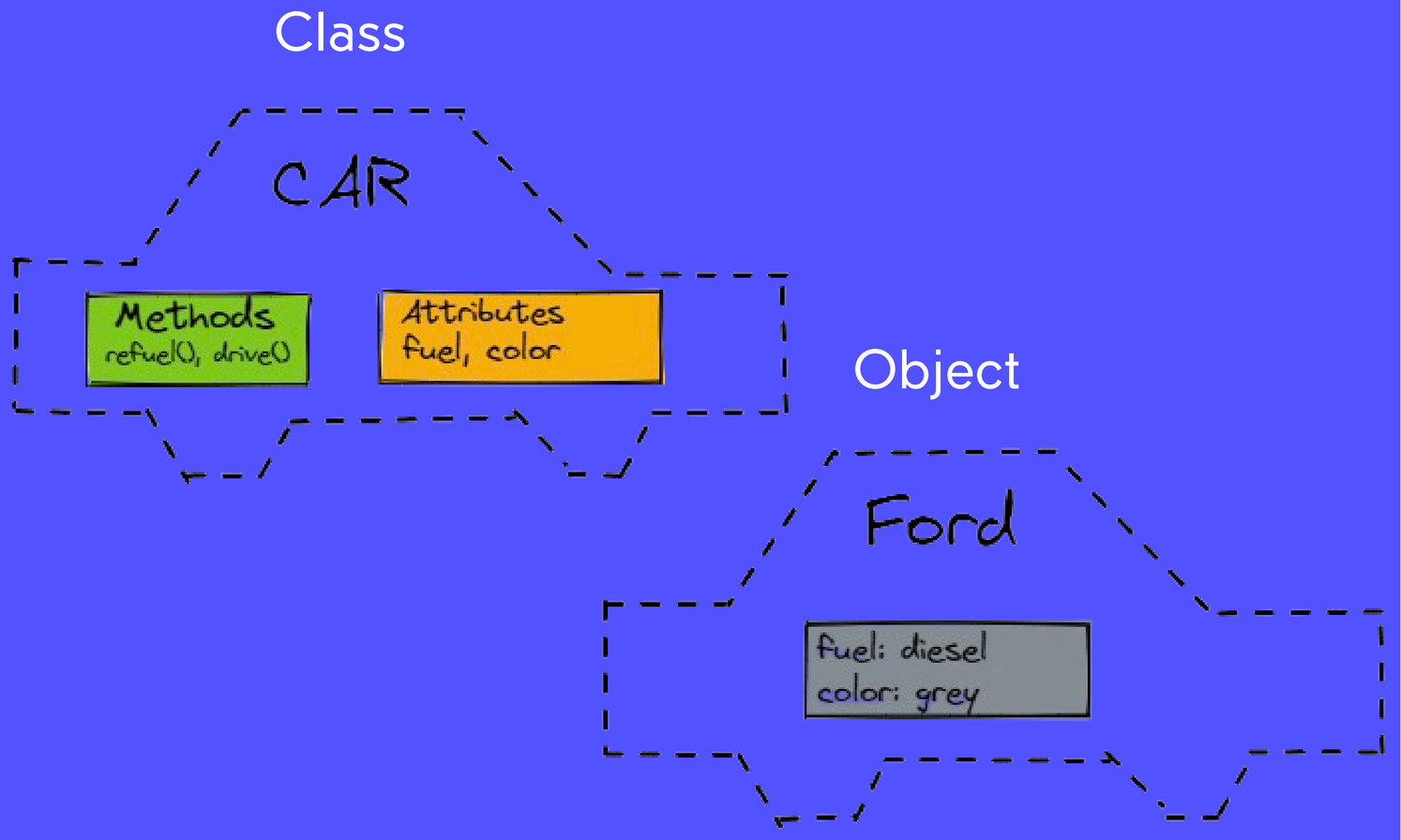
Primitive Data Types

- Basic **building blocks** of data
- **Cannot** be broken down into simpler structures of data
- In Javascript, there are 7 primitive data types: **string, number, bigint, boolean, undefined, symbol and null**
- For example, a JS Array is **not** primitive because it can be **broken down** into each of its elements and manipulated



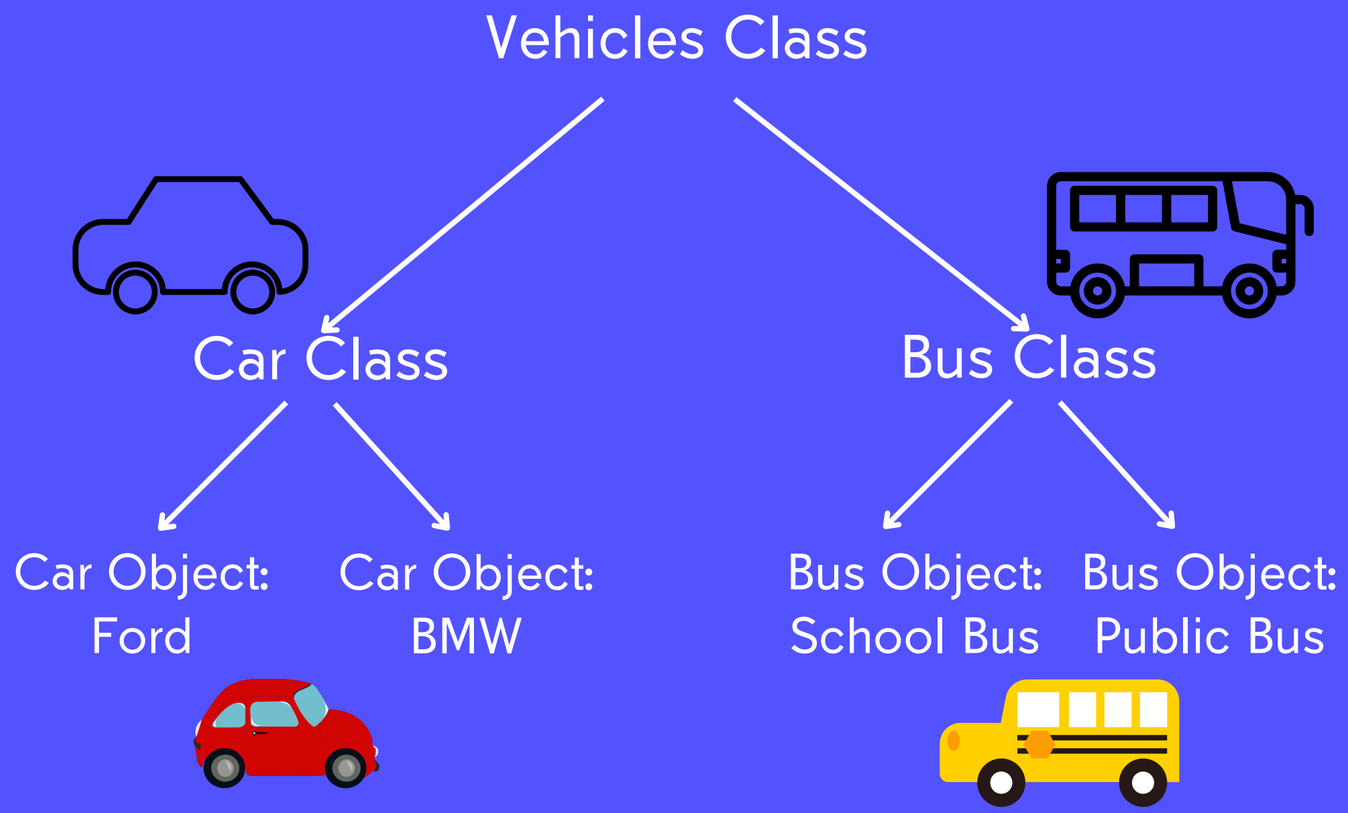
User Defined Types

- Allow users to create their own data types, with classes and create instances, objects
- Can add **attributes**, which is a variable containing data, to the data type that can be manipulated
- Or add **methods** if you want functions to be used on the object



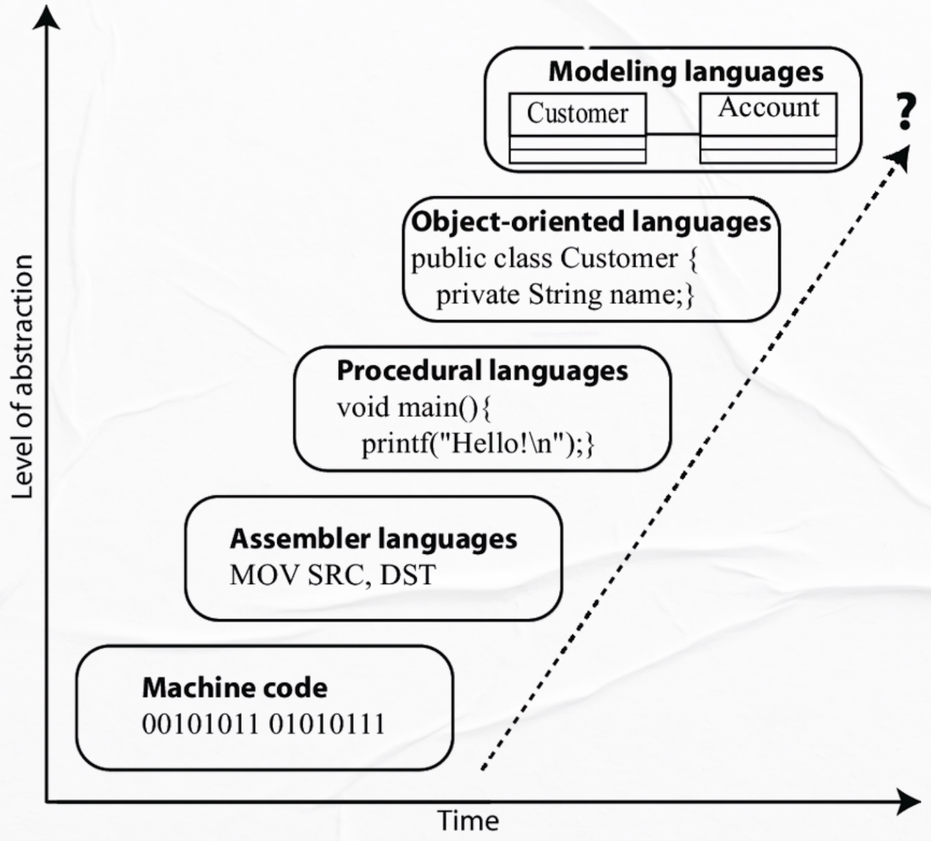
User Defined Types Cont'd

- Plus, allows **inheritance**, meaning when a class is extended to another class, it is created with **all the predefined attributes of its parent class**
- Means an instance of that class will have the type of both classes
- Concept of object having multiple types -> known as **polymorphism**



Abstraction

Abstraction in computer programming is when the complexity of a system is hidden beneath a high level idea



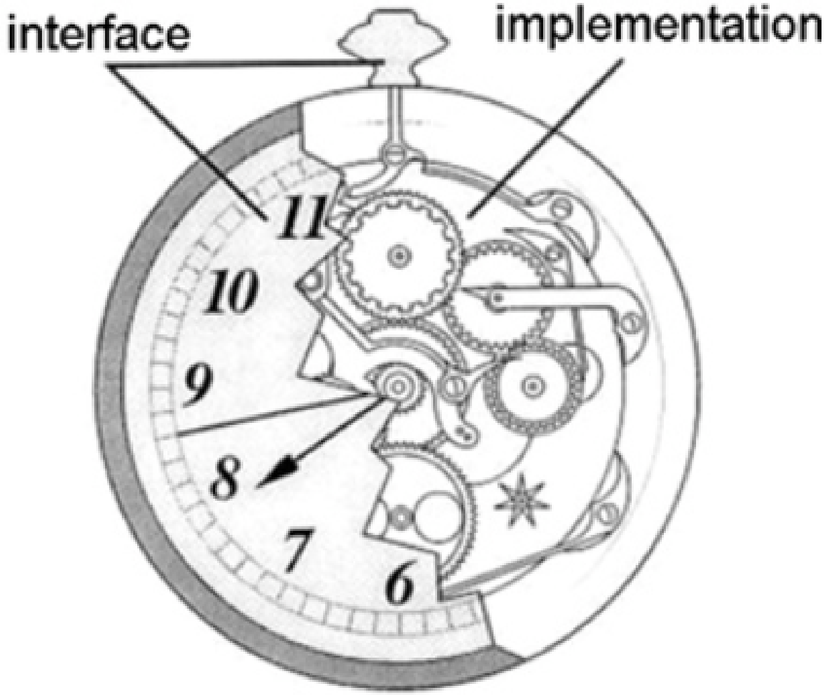
Interface vs Implementation

Interface

- Interface is the **layer of abstraction** that is used by the user to code

Implementation

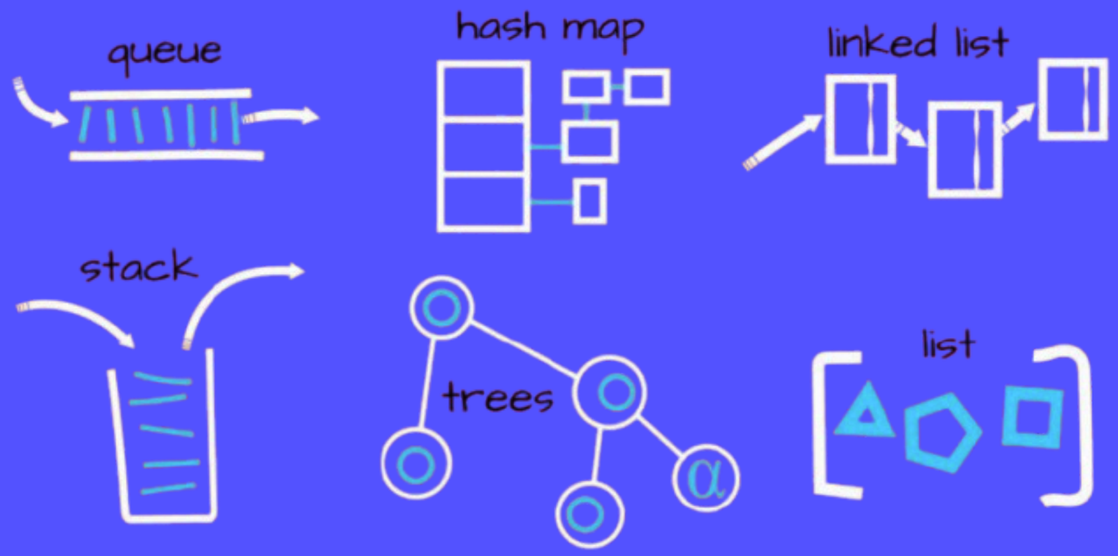
- Implementation is the actual code behind the layer of abstraction that runs to make the code execute its function
- With primitive data types, there is almost no interface, **mostly implementation**, since you **directly** code the operations performed on them usually



What are ADTs?

They are high level interfaces that can be implemented with other data types

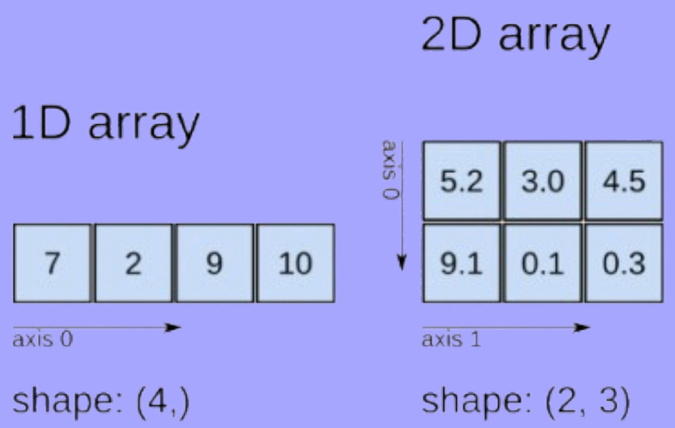
- It specifies more of the **logicalities, operations and behavior of a data structure** without **restricting** it to certain data types and going into **detail** of how those behaviors are **implemented**
- They can perform basic CRUD operations, but also include ADT specific operations such as
  - Traverse
  - Search
  - Inserting
  - Sorting



ADTs vs Data Structures

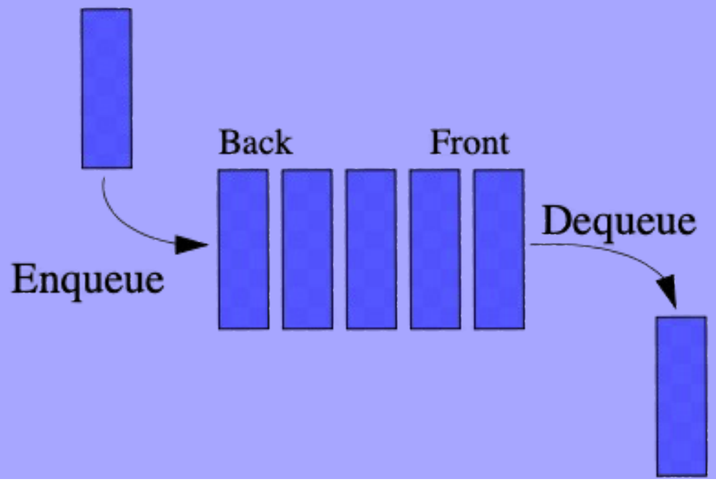
ADTs

- They rely on an a data structure to implement the methods part of their interface
- ADTs are more high level
- Data items are in logical form
- Specify the operations and behaviour



Data Structures

- Data structures involve handling the storage format of the data
- Data items are in a physical form
- Can **implement ADTs** to specify behaviour of that data structure

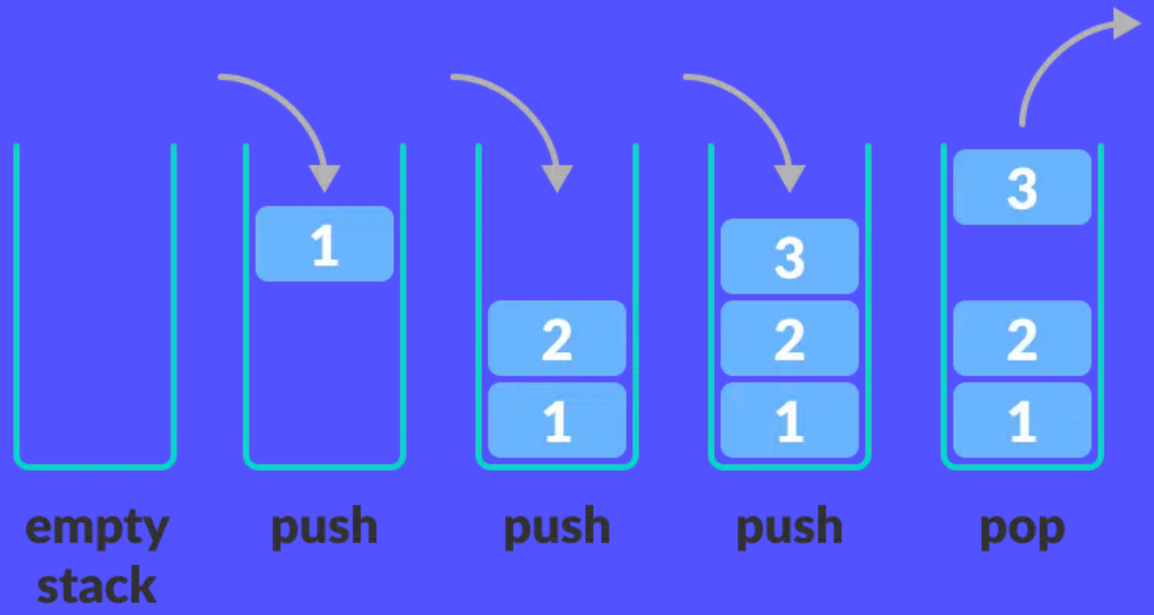


Stacks

Structured as an order collection of items where items are added and removed from the end called the “top”. Stacks are LIFO (last in, first out).

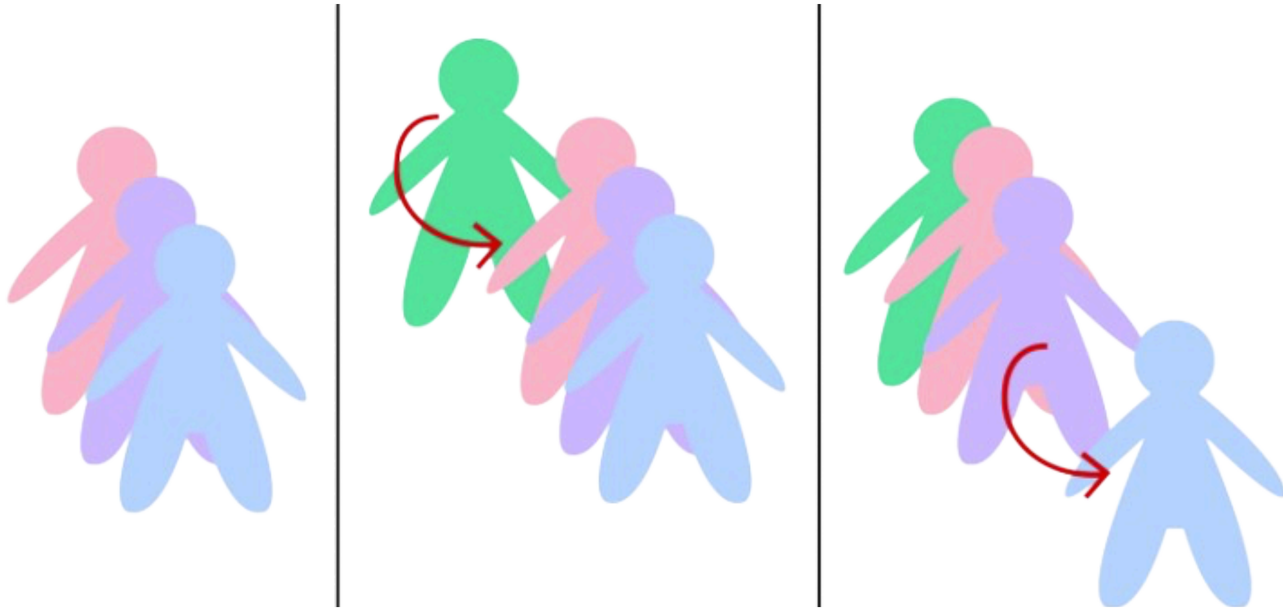
Stack-specific operations

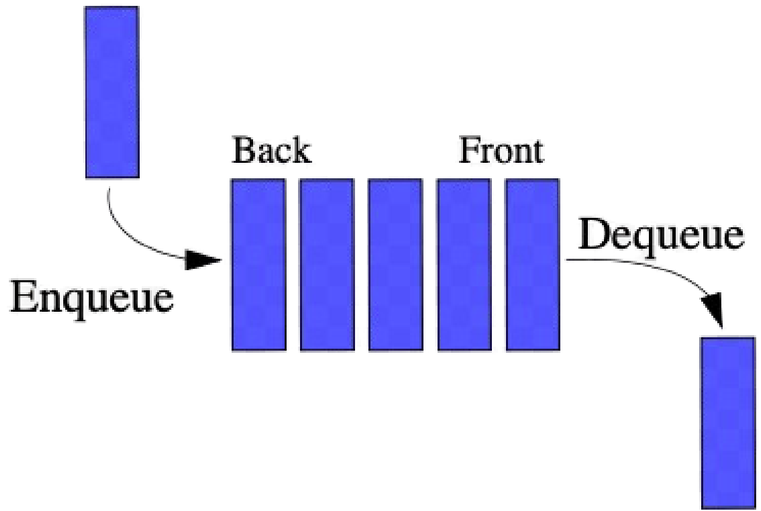
- **Push:** adds new item to the top of the stack
- **Pop:** removes top item from stack
- **Peek:** returns current top item of stack



Queues

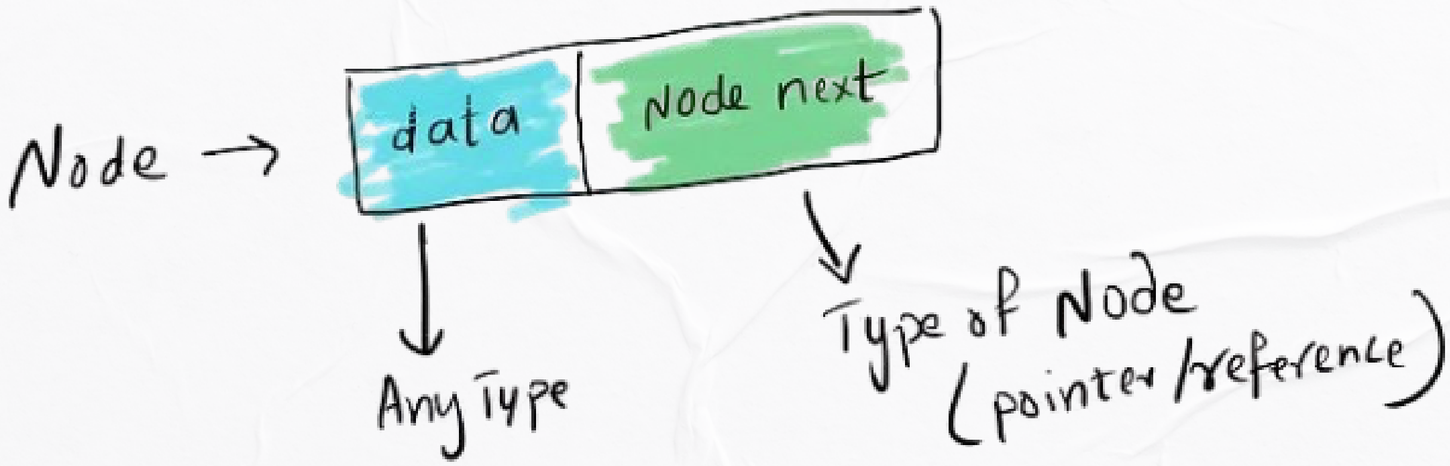
- Collection where elements are added at one end (rear or tail) and removed from the other end (front or head)
- Queue is a **FIFO (first in, first out)**
- **Cannot** directly access middle elements of a queue
- Queue-specific operations:
  - **Enqueue:** adds element to the rear of the queue
  - **Dequeue:** removes element from front of the queue
  - **First:** returns first element at the front of the queue
  - **toString:** returns string representation of the queue





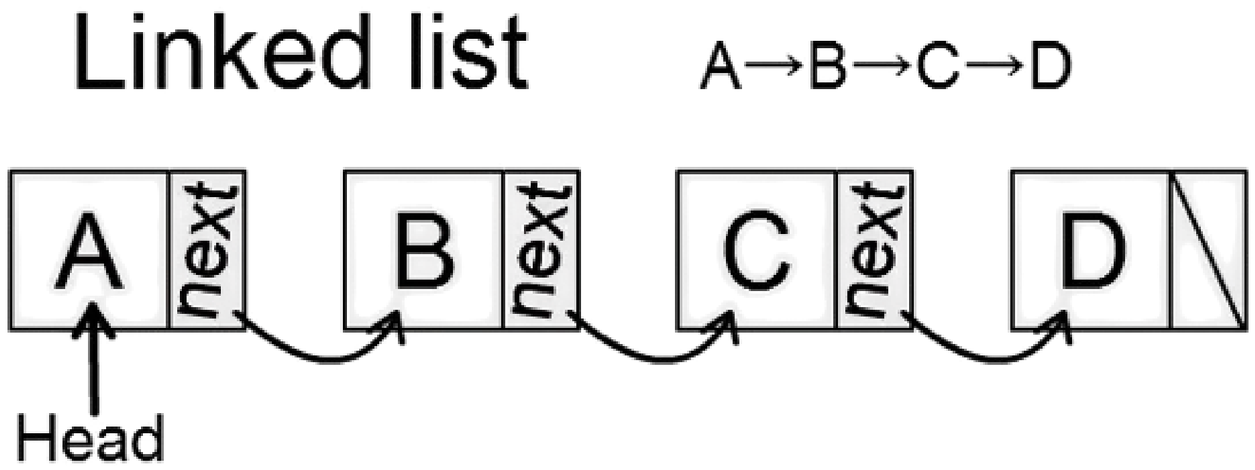
Nodes

- Building blocks for any data structure and ADT
- Have a value and pointer
- Value -> any data type
- Pointer -> points to where next node structure is



Linked Lists

Basic structure of linked nodes. 1st node points to 2nd and so on. Nodes can be implemented with any data type, from primitive to user-defined



Operations

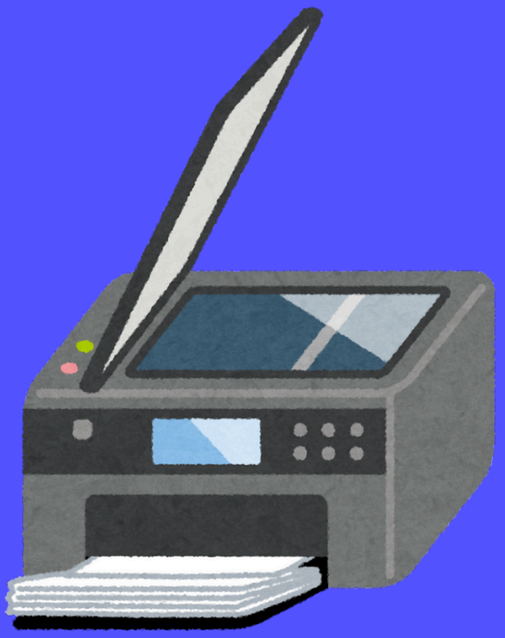
- **Prepend:** add a node to the beginning
- **Append:** add a node to the end
- **Pop:** remove the node from the end
- **popFirst:** remove the node from the beginning
- **Head:** return first node
- **Tail:** return last node
- **Remove:** remove node from the list

Why use ADTs?

- Encapsulation
  - Will provide certain methods and attributes. User only needs to know them to use the ADT
- Compartmentalization
  - Code using ADT won’t be impacted if changes are made to how the implementation part of the ADT works
- Adaptability
  - When real world situations and its requirements evolve, ADTs can easily adapt to the situation in which they’re used.

Real World Applications

- Undo/redo mechanisms in text and photo editors, utilises stack ADT
- Task scheduling & print job scheduling, utilises queue ADT
- Music playlists and dynamic memory allocation, utilises linked lists ADT to manage songs and efficient memory usage, respectively



Thank You

Presentation by **Haya R**