Interfaces with Example

**What Is an Interface?**

As you've already learned, objects define their interaction with the outside world through the methods that they expose. Methods form the object's *interface* with the outside world; the buttons on the front of your television set, for example, are the interface between you and the electrical wiring on the other side of its plastic casing. You press the "power" button to turn the television on and off.

In its most common form, an interface is a group of related methods with empty bodies. A bicycle's behavior, if specified as an interface, might appear as follows:

interface Bicycle {

// wheel revolutions per minute

void changeCadence(int newValue);

void changeGear(int newValue);

void speedUp(int increment);

void applyBrakes(int decrement);

}

To implement this interface, the name of your class would change (to a particular brand of bicycle, for example, such as ACMEBicycle), and you'd use the implements keyword in the class declaration:

class ACMEBicycle **implements** Bicycle {

int cadence = 0;

int speed = 0;

int gear = 1;

// The compiler will now require that methods

// changeCadence, changeGear, speedUp, and applyBrakes

// all be implemented. Compilation will fail if those

// methods are missing from this class.

void changeCadence(int newValue) {

cadence = newValue;

}

void changeGear(int newValue) {

gear = newValue;

}

void speedUp(int increment) {

speed = speed + increment;

}

void applyBrakes(int decrement) {

speed = speed - decrement;

}

void printStates() {

System.out.println("cadence:" +

cadence + " speed:" +

speed + " gear:" + gear);

}

}

Implementing an interface allows a class to become more formal about the behavior it promises to provide. Interfaces form a contract between the class and the outside world, and this contract is enforced at build time by the compiler. If your class claims to implement an interface, all methods defined by that interface must appear in its source code before the class will successfully compile.

Why use interface ?

Efficiency and Better implementation of a class or process

**Linked List**

In [computer science](https://en.wikipedia.org/wiki/Computer_science), a **linked list** is a linear collection of data elements, called nodes, each pointing to the next node by means of a [pointer](https://en.wikipedia.org/wiki/Pointer_(computer_programming)). It is a [data structure](https://en.wikipedia.org/wiki/Data_structure)consisting of a group of [nodes](https://en.wikipedia.org/wiki/Node_(computer_science)) which together represent a [sequence](https://en.wikipedia.org/wiki/Sequence). Under the simplest form, each node is composed of data and a [reference](https://en.wikipedia.org/wiki/Reference_(computer_science)) (in other words, a *link*) to the next node in the sequence. This structure allows for efficient insertion or removal of elements from any position in the sequence during iteration. More complex variants add additional links, allowing efficient insertion or removal from arbitrary element references.

The principal benefit of a linked list over a conventional [array](https://en.wikipedia.org/wiki/Array_data_structure) is that the list elements can easily be inserted or removed without reallocation or reorganization of the entire structure because the data items need not be stored contiguously in memory or on disk, while an array has to be declared in the source code, before compiling and running the program. Linked lists allow insertion and removal of nodes at any point in the list, and can do so with a constant number of operations if the link previous to the link being added or removed is maintained during list traversal.

## **Advantages[**[**edit**](https://en.wikipedia.org/w/index.php?title=Linked_list&action=edit&section=1)**]**

* Linked lists are a dynamic data structure, which can grow and be pruned, [allocating and deallocating memory](https://en.wikipedia.org/wiki/Allocating_and_deallocating_memory) while the program is running.
* Insertion and deletion node operations are easily implemented in a linked list.
* Linear data structures such as stacks and queues can be implemented using a linked list.
* There is no need to define an initial size for a Linked list.
* Items can be added or removed from the middle of list.

## **Disadvantages[**[**edit**](https://en.wikipedia.org/w/index.php?title=Linked_list&action=edit&section=2)**]**

* They use more memory than [arrays](https://en.wikipedia.org/wiki/Array_data_structure) because of the storage used by their [pointers](https://en.wikipedia.org/wiki/Pointer_(computer_science)).
* Nodes in a linked list must be read in order from the beginning as linked lists are inherently [sequential access](https://en.wikipedia.org/wiki/Sequential_access).
* Nodes are stored incontiguously, greatly increasing the time required to access individual elements within the list, especially with a [CPU cache](https://en.wikipedia.org/wiki/CPU_cache).
* Difficulties arise in linked lists when it comes to reverse traversing. For instance, singly linked lists are cumbersome to navigate backwards[[1]](https://en.wikipedia.org/wiki/Linked_list#cite_note-1) and while doubly linked lists are somewhat easier to read, memory is wasted in allocating space for a [back-pointer](https://en.wikipedia.org/wiki/Back-pointer).

Each record of a linked list is often called an 'element' or '[node](https://en.wikipedia.org/wiki/Node_(computer_science))'.

The field of each node that contains the address of the next node is usually called the 'next link' or 'next pointer'. The remaining fields are known as the 'data', 'information', 'value', 'cargo', or 'payload' fields.

### Singly linked list

Singly linked lists contain nodes which have a data field as well as a '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.

[ingly-linked-list.svg](https://en.wikipedia.org/wiki/File:Singly-linked-list.svg)  
*A singly linked list whose nodes contain two fields: an integer value and a link to the next node*

### 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 'forward('s') and 'backwards', or 'next' and 'prev'('previous').

[oubly-linked-list.svg](https://en.wikipedia.org/wiki/File:Doubly-linked-list.svg)

### Circular Linked list

In the last [node](https://en.wikipedia.org/wiki/Node_(computer_science)) of a list, the link field often contains a [null](https://en.wikipedia.org/wiki/Null_pointer#Null_pointer) reference, a special value 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'.

[ircularly-linked-list.svg](https://en.wikipedia.org/wiki/File:Circularly-linked-list.svg)

talk about implements ( interface in class) extends( inheritance in class). A java class cannot extend more than one class, so additional functionality can be incorporated by using interfaces.

But a java class can implement as many number of interfaces

Deadly diamond of dead problem