When were called, generator functions do not initially execute their code. Instead they return a special type of iterator, called a Generator.

When a value is consumed by calling the generator's next method the Generator function executes until it encounter the yield keyword.

→ The execution of the generator function doesn't start at the time of calling the function.

→ The moment we get yield, the execution of the generator function stop there. and whatever we was yielding, it returns till that value.

→ 'yield' is similar to return but not a return

→ We can even pass a value to the iter.next() that is going to place that value at the same position where you have last yielded yourself.

→ If we don't yield, then it will search for return and if we don't have any return, it will by default return undefined.

* Write async in front of the function, it denotes that, for this function has some asynchronous activity. So it will start pointing like asynchronous javascript, and consume these promises with await.

* **Async and Await :—**

It provides a new way to write asynchronous code that is easier to read, write and debug than traditional callback-based approaches.

Async/Await is built on top of promises. A promise is an object that represents the eventual completion or failure of an asynchronous operation and allows you to attach callbacks to handle the outcome.

Async/Await allows us to write asynchronous code as if it were synchronous by using the "await" keyword to pause the execution of the code until a promise is resolved.

* **catch () :—**

In JS, "catch" is a method that is used to handle errors that may occur during the execution of a Promise. The "catch" method is typically chained onto to the end of a promise chain and takes a single argument, which is a function that will be called if an error occurs at any point in the promise chain.

Example

```
somePromise()
.then((result) => {
    console.log(result);    // Handle the successfull outcome
})
.catch((error) => {
    // Handle the failure outcome
    console.log(error);
})
```

Date / /

* try... catch

The "try... catch" statement is comprimised of a try block and either a catch block, a finally block, or both.

The code in the try block executed first, and if it throws an exception, the code in the catch block will be executed. The code in the finally block will always be executed before control flow exits the entire construct.