Symbol

→ What is the symbol ?

→ A symbol is a primitive which cannot be recreated. It was introduced in ECMA Script 2015. It is a very peculiar data type. Once we create a symbol, its value is kept private and for internal use. All that remains after the creation of the symbol is the symbol reference. We can create a symbol by calling the Symbol() global factory function:

```
const mySymbol = Symbol()
```

Every time we invoke Symbol(), we get a new and unique symbol, which is different from all other symbols:

```
const sym1 = Symbol()
const sym2 = Symbol();
console.log(s1 === s2); // FALSE
```

We can pass a parameter to Symbol(), and that is used as the symbol description, useful just for debugging purposes:

```
console.log(Symbol()) // Symbol()
console.log(Symbol('some text')) // Symbol(some text)
```

EXAMPLE:

```
const mySymbol = Symbol()
const person = {
    [mySymbol]: 'John'
};
person[mySymbol]
const mySymbol2 = Symbol()
person[mySymbol2] = () => 'Hello World'
console.log(person[mySymbol2]())
// OUTPUT: Hello World
```

Page No.
Date

→ Purpose of the symbol:

→ In ES6, symbols were added to the primitives group, just like all other primitive, they are also immutable and doesn't have methods of their own. The main purpose of symbol was to provide globally unique values that were kept private and for internal use only.

→ Use Case: Symbols as keys of non-public properties:

When there are inheritance heirarchies in JavaScript, we have two kinds of properties. First one is created via classes, and the second is a purely prototypal approach:

→ Public properties: They are seen by clients of the code.

→ Private properties: They are used internally within the pieces that make up the inheritance heirarchy such as classes, objects.

For usability, public properties usually have string keys. However, in the case of private properties with string keys, accidental name clashes can become a problem. Therefore, symbols are the right choice. Here is an example:

Page No.
Date

Suppose that there are two companies, A and B, developed by two different people. Both of them need to add a property to work on an object, but both end up naming their property id by coincidence. This coincidence leads to one of the companies overwriting the data stored in id. Before ES6, when the object key could only be of the string type, this scenario was a real possibility. String type object keys paved the danger of name collision and led to the overwriting of data/values. In such a scenario, symbols have provide a solution to this problem.

```
LET STUDENT = { NAME: "HARRY" };
LET ID_COMPANY_A = SYMBOL("ID");
STUDENT[ID_COMPANY_A] = "ID ASSIGNED BY COMPANY A";
LET ID_COMPANY_B = SYMBOL("ID");
STUDENT[ID_COMPANY_B] = "ID ASSIGNED BY COMPANY B";
CONSOLE.LOG(STUDENT)
// Output: { NAME: "HARRY", SYMBOL(ID): "ID ASSIGNED BY COMPANY A", SYMBOL(ID): "ID ASSIGNED BY COMPANY B" }
```

→ Conclusion: Symbols in JavaScript can provide uniqueness to objects. It is worthwhile for all developers to have a basic understanding

Date

of them and their various use-cases.