IIFE:

- define an anonymous function and then immediately run that function as soon as it has been defined. In JavaScript we call this an Immediately-Invoked Function Expression or IIFE (pronounced as “iffy”).
- A function expression is when you define a function and assign that function to a variable:

```
// here we are assigning a named function declaration to a variable
const sayHi = function sayHello() {
  console.log("Hello, World!");
};

sayHi(); // prints "Hello, World!"
```

- We can also use function expression syntax to assign variables to anonymous functions effectively giving them names:

```
// here we are assigning an anonymous function declaration to a variable
const sayHi = function() {
  console.log("Hello, World!");
};

sayHi(); // prints "Hello, World!"
```

- Now what if we only ever wanted to **invoke the above anonymous function once without assigning it a name?**

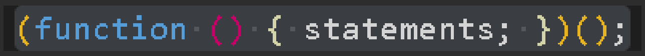
- To do that we can define an Immediately-Invoked Function Expression.

IIFE syntax

An *Immediately-Invoked Function Expression* is a function that is called immediately after it had been defined. The typical syntax we use to write an IIFE is to start by writing an anonymous function and then wrapping that function with the grouping operator, `( )`. After the anonymous function is wrapped in parenthesis you simply add another pair of closed parenthesis to invoke your function.

Here is the syntax as described:

```
// 1. wrap the anonymous function in the grouping operator
// 2. invoke the function!
(function() {
  statements;
})();
```



```
(function () { statements; })();
```

Let's take a look at an example. The below function will be invoked right after it has been defined:

```
(function() {
  console.log("run me immediately!");
})();
// => 'run me immediately!'
```

This function will be defined, invoked, and then will *never be invoked again*. What we are doing with the above syntax is forcing JavaScript to run our function as a **function expression** and then to invoke that function expression immediately.

Since an Immediately-Invoked Function Expression is *immediately invoked* attempting to assign an IIFE to a variable will return the value of the invoked function.

Here is an example:

```
let result = (function() {
  return "party!";
})();

console.log(result); // prints
"party!"
```

So we can use IIFEs to run an anonymous function immediately and we can:

still hold onto the result of that function by assigning the IIFE to a variable.

IIFEs keep functions and variables private

- Using IIFEs ensures our global namespace remains unpolluted by a ton of function or variable names we don't intend to reuse.

- IIFEs can additionally protect global variables to ensure they can't ever be read or overwritten by our program. When learning about scope we talked about how an outer scope does not have access to an inner scope's variables:

```
function nameGen() {
  const bName = "Barry";
  console.log(bName);
}

// we can not reference the bName variable from this
outer scope
console.log(bName);
```

- Now what if we didn't want our outer scope to be able to access our function at all?
- Say we wanted our `nameGen` function to only be invoked once and not ever be invoked again or even to be accessible by our application again?
- One of the main advantages gained by using an IIFE is the very fact that the function cannot be invoked after the initial invocation. Meaning that no other part of our program can ever access this function again.

Let's take a look at rewriting our nameGen function using a sneaky IIFE:

```
(function() {
  const bName = "Barry";
  console.log(bName);
})(); // prints "Barry"

// we still cannot reference the bName variable from this outer
scope
// and now we have no hope of ever running the above function
above again

console.log(bName);//
```

- Since we don't ever intend to invoke this function again - there is no point in assigning a name to our function.

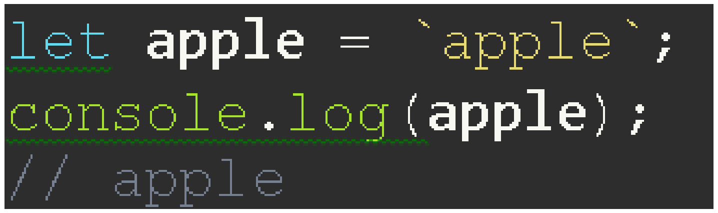
Interpolation in JavaScript

- **A Template Literal is a new way to create a string literal that expands on the syntax of the String primitive type allowing for interpolated expressions to be inserted easily into strings.**

Let's talk syntax:

- To create a template literal, instead of single quotes (`'`) or double quotes (`"`) we **use the grave character**:
- **also known as the backtick (`).**

Defining a new string using a template literal :



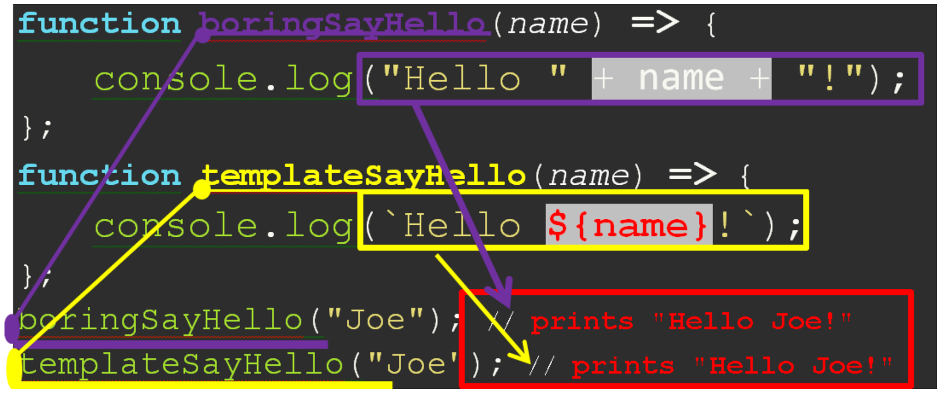
```
let apple = `apple`;
console.log(apple);
// apple
```

- we can simply use backticks to create that new string:
- The important thing to know is that **a template literal is still a `String`** - just with some really nifty features!

Interpolation using template literals

- One of the main advantages we gain by using template literals is the ability to **interpolate *variables or expressions into strings.***
  - We do this by denoting the ***values we'd like to interpolate by wrapping them within curly braces with a dollar sign in front(`${value_we_wish_to_interpolate}`).***
  - When your code is being run :
    - the variables or expressions wrapped within the `${}` will be evaluated
    - then will be replaced with the value of that variable or expression.

Compare how easy to read the following two functions are:



```
function boringSayHello(name) => {
    console.log("Hello " + name + "!");
};
function templateSayHello(name) => {
    console.log(`Hello ${name}!`);
};
boringSayHello("Joe"); // prints "Hello Joe!"
templateSayHello("Joe"); // prints "Hello Joe!"
```

As we can see in this example:

- the value of the variable is being interpolated into the string that is being created using the template literal.
- This makes our code easier to both write and read.

You'll most often be interpolating variables with template literals, however we **can also interpolate expressions**.

```
let string = `Let me tell you ${1 + 1} things!`;
console.log(string);
// Let me tell you 2 things!
```

Here is an example of
evaluating an expression within a template literal:

Multi-line strings using template literals:

- **We can also use template literals to create multi-line strings!**
  - **Previously we'd write multi-line strings by doing something like this:**

```
console.log("I am an example\n" + "of an extremely long string");
```

- **Using template literals this becomes even easier:**

```
console.log(`I am an example
of an extremely long string
on multiple lines`);
```

Object Keys and Symbols

- Up to this point we have been using strings as our object `keys`.
- However, as of ES6 edition of JS we can use another data type to create Object keys: `Symbols`.
- an object's keys can be either a `String` *or* a `Symbol`.

A symbol of unique freedom

Let's look at the syntax used to create a new symbol:

The main advantage of using the ***immutable Symbol primitive data type*** is that : ***each Symbol value is unique***.

- Once a symbol has been created it **cannot** be recreated.

Keyword: ***Symbol***

```
// the description is an optional string used for debugging
Symbol([description]);
```

```
const sym1 = Symbol();
console.log(sym1); // Symbol()
```

To create a new symbol you call the Symbol() function which will return a new unique symbol:

Here is an example of how each created symbol is guaranteed to be **unique**:

```
const sym1 = Symbol();
const sym2 = Symbol();

console.log(sym1 === sym2); // false
```

You can additionally pass in an optional description string that will allow for easier debugging by giving you an identifier for the symbol you are working with:

```
const symApple = Symbol "apple"
console.log(symApple); // Symbol(apple)
```

```
const symApple1 = Symbol('apple");
const symApple2 = Symbol('apple");

console.log(symApple1); // Symbol(apple)
console.log(symApple2); // Symbol(apple)
console.log(symApple1 === symApple2); // false

Symbol("foo") === Symbol("foo"); // returns false
```

- Don't confuse the optional description string as way to access the Symbol you defining though
  - the string value isn't included in the Symbol in anyway.
- We can invoke the `Symbol()` function multiple times with the same description string and:
-
  - each newly returned symbol will be unique:

Symbols vs. Strings as keys in Objects:

The main use of `Symbol` primitive data type (JS):
to create unique object keys.

- ***Here is a brief refresher on that syntax:***

```
const dog = {};
dog["sound"] = "woof";
dog.age = 4;
// using bracket notation with a variable
const col = "color";
dog[col] = "grey";
console.log(dog);
 // { sound: 'woof', age: 4, color: 'grey' }
```

- One of the problems with using strings as object keys is that
  - in the **Object type: each key is *unique*.**
- **could inadvertently overwrite a key's value if we mistakenly**:
  - **reassign** the **same key name twice:**

```
const dog = {};
// I set an 'id' key value pair on my dog
dog["id"] = 39;
---------------------------------------------------
// Here imagine someone else comes into my code base and
// accidentally adds another key with the same name!
dog.id = 42;
console.log(dog);//  { id: 42 }
```

- When a computer program attempts to use the same variable name twice for different values we call this a ***name collision***.
- The more libraries that are imported into a single application the greater the chance of a name collision.

- Using `Symbols` as your object keys is a great way to prevent name collision on objects **because Symbols are *unique*!**

- **Let take a took at how we could fix the above code snippet using symbols:**

```
const dog = {};
const dogId = Symbol("id");
dog[dogId] = 39;

const secondDogId = Symbol("id");
dog[ secondDogId  ] = 42;

console.log(dog[dogId]); // 39
console.log(dog);
//{[Symbol(id)]: 39,[Symbol(id)]: 42}
```

- Note above that we can access our key value pair using bracket notation …
- and passing in the variable we assigned our symbol to:
  - **(in this case…*`dogId`*).**

Iterating through objects with symbol keys

One of the other ways that `Symbols` differ from `String` keys in an object is the way we iterate through an object. Since `Symbols` are relatively new to JavaScript older Object iteration methods don't know about them. This includes using **`for...each` and `Object.keys()`:**

```
const name = Symbol("name");
const dog = {
  age: 29
};
dog[name] = "Fido";
console.log(dog);
// { age: 29, [Symbol(name)]: 'Fido' }
console.log(Object.keys(dog)); // prints ["age"]
for (let key in dog) {
  console.log(key);
} // prints age
```

This provides an additional bonus to using symbols as object keys because your symbol keys that much more hidden (and safe) if they aren't accessible via the normal object iteration methods.

```
const name = Symbol("name");
const dog = {
  age: 29,
//when defining an object
//we can use square brackets within an object
//To interpolate a variable key
  [name]: "Fido"
};
Object.getOwnPropertySymbols(dog);
// prints [Symbol(name)];
```

If we do want to access all the symbols in an object we can use the built in `Object.getOwnPropertySymbols()`.

Using symbols as object keys has some advantages in your local code but they become really beneficial when dealing with importing larger libraries of code into your own projects.

The more code imported into one place means the more variables floating around which leads to a greater chance of a name collisions - which can lead to some really devious debugging.

Primitive Data Types in Depth

Data types in JavaScript

- **With the JavaScript ECMAScript 2015 release, there are now eight different data types in JavaScript.**
    - **There are seven primitive types and one reference type.**

Primitive Types:

1. `Boolean` - `true` and `false`
2. `Null` - represents the intentional absence of value.
3. `Undefined` - default return value for many things in JavaScript.
4. `Number` - like the numbers we usually use (`15`, `4`, `42`)
5. `String` - ordered collection of characters ('apple')
6. `Symbol` - new to ES5 a symbol is a *unique* primitive value
7. `BigInt` - a data type that can represent larger integers than the `Number` type can safely handle.

- main differences between primitive and reference data types in JavaScript:
    - primitive data types are **immutable**, meaning that they cannot be changed.

One Reference Type:

1. `Object` - a collection of properties and methods

- The other main thing that sets primitives apart is that primitive data types are not Objects and therefore *do not have methods*.

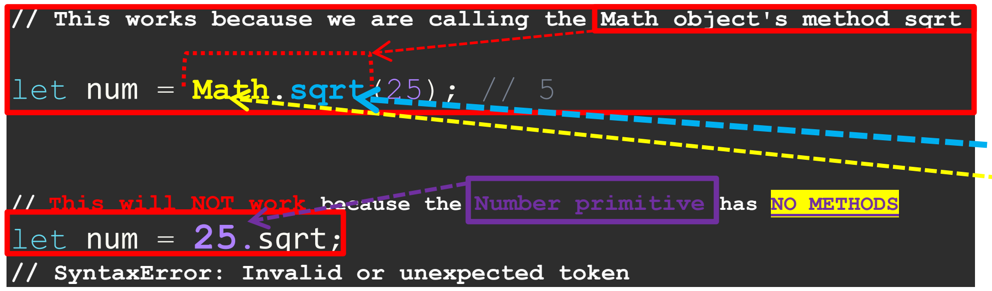
Methods and the object type

the definition of a method is a function that belongs to an object.

```
const corgi = {
  name: "Ein",
  func: function() {
    console.log("This is a method!");
  }
};
corgi.func();
// prints "This is a method!"
```

- The other main thing that sets primitives apart is that primitive data types are not Objects and therefore *do not have methods*.
- The Object type is the only data type in JavaScript that has methods. **Meaning that primitive data types cannot have methods**

For example when finding the square root of a number in JavaScript we would do the following:



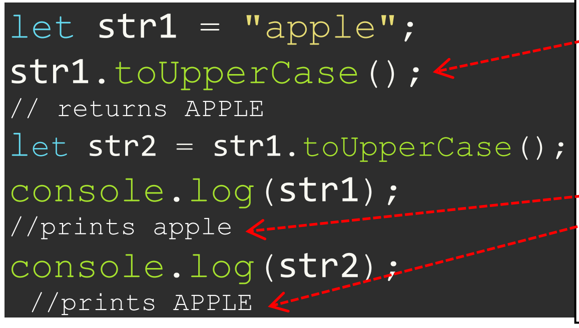
```
// This works because we are calling the Math object's method sqrt
let num = Math.sqrt(25); // 5

// This will NOT work because the Number primitive has NO METHODS
let num = 25.sqrt;
// SyntaxError: Invalid or unexpected token
```

- The `Number` primitive data type above (`25`) does not have a `sqrt` method because:
- only Objects in JavaScript can have methods.

Primitives with object wrappers

- The underlying primitive data type of `String` does not have methods.
- To make the String data type easier to work with in JavaScript it is implemented using a `String` object that *wraps* around the `String` primitive data type.
- This means that the `String` object will be responsible for constructing new `String` primitive data types as well as allowing us to call methods upon it because it is an Object.
  - ***The difference between the `String` primitive type, and the `String` object that wraps around it:***



```
let str1 = "apple";
str1.toUpperCase();
// returns APPLE
let str2 = str1.toUpperCase();
console.log(str1);
//prints apple
console.log(str2);
 //prints APPLE
```

- When we call `str1.toUpperCase()` we are calling that on the `String` object that wraps around the `String` primitive type.
- This is why in the above example when we `console.log` both `str1` and `str2` we can see they are different.
- The value of `str1` has not changed because it can't - the `String` primitive type is *immutable*.
- The `str2` variable calls the `String#toUpperCase` method on the `String` object which wraps around the `String` primitive

- This method cannot mutate the `String` primitive type itself - it can only point to a new place in memory where the `String` primitive for `APPLE` lives.
- This is why even though we call `str1.toUpperCase()` multiple times the value of that variable will never change until we reassign it.

Unassigned Variables in JavaScript

The default value of variables

Whenever you declare a `let` or `var` variable in JavaScript that variable will have:

a *name* and a *value*.

- That is true of `let` or `var` variables with no assigned value.
- When declaring a variable using `let` or `var` we can optionally assign a value to that variable.

Let's take a look at an example of declaring a variable with `var`:

```
var hello;
console.log(hello);
// prints undefined
```

- So when we declare a variable using `var` the default value assigned to that variable will be `undefined` (if no value is assigned.)

```
let goodbye;
console.log(goodbye);
// prints undefined
```

- The same is true of declaring a variable using `let`.
- When declaring a variable using `let` we can also choose to optionally assign a value:

- However, this is a case where `let` and `const` variables differ.
  - **When declaring a variable with `const` we must provide a value for that variable to be assigned to.**

```
const goodbye;
console.log(goodbye;

//SyntaxError:
//Missing initializer in const
declaration
```

- This behavior makes sense because a `const` variable cannot be reassigned
- - meaning that is we don't assign a value when a `const` variable is declared we'd never be able to assign a value!

- So the default assigned value of a variable declared using `var` or `let` is `undefined`,
  - whereas variables declared using `const` need to be assigned a value.

The difference between default values and hoisting

- **When talking about default values for variables we should also make sure to emphasize the distinction between *hoisting* variable names and default values.**

```
function hoistBuddy() {
  var hello;
  console.log(hello);
// prints undefined
}
hoistBuddy();
```

- Whenever a variable is declared with `var` that variable's name is hoisted to the top of its declared scope with a value of `undefined` until a value is assigned to that variable name.
  - If we never assign a value to the declared variable name then the default value of a var declared variable is undefined(w/o assignment).

- **Now let's take a look at the example above but this time using `let`:**

```
function hoistBuddy() {
  let hello;
  console.log(hello);
 // prints undefined
}

hoistBuddy();
```

- The default value of a **`let`** declared variable is `undefined`.
- When a **`let`** variable is declared the name of that variable is hoisted to the top of its declared scope and if a line of code attempts to interact with that variable before it has been assigned a value .....an error is thrown.

- **The following example shows the difference in** hoisting **between `var` and `let` declared variables:**

```
function hoistBuddy() {
  console.log(hello);
// prints undefined
  var hello;


  console.log(goodbye);
// ReferenceError:
Cannot access 'goodbye' before initialization
  let goodbye;
}
hoistBuddy();
```

What you learned

- The default value of a variable assigned with either `let` or `var` is `undefined`.
- Variables declared using `const` need to have an assigned value upon declaration to be valid.