### Learning Advanced JavaScript

Defining Functions

Named Functions

Functions as Objects

Instantiation

Flexible Arguments

Closures

Temporary Scope

Function Prototypes

Instance Type

Inheritance

Built-in Prototypes

Enforcing Function Context

Bonus: Function Length (Overloading , and Overriding)

### Goal: To be able to understand this function:

// The .bind method from Prototype.js   
Function.prototype.bind = function(){   
  var fn = this, args = Array.prototype.slice.call(arguments), object = args.shift();   
  return function(){   
    return fn.apply(object,   
      args.concat(Array.prototype.slice.call(arguments)));   
  };   
};

### Some helper methods that we have:

assert( true, "I'll pass." );   
assert( "truey", "So will I." );   
assert( false, "I'll fail." );   
assert( null, "So will I." );   
log( "Just a simple log", "of", "values.", true );   
error( "I'm an error!" );

### What ways can we define functions?

function isNimble(){ return true; }   
var canFly = function(){ return true; };   
window.isDeadly = function(){ return true; };   
log(isNimble, canFly, isDeadly);

### Does the order of function definition matter?

var canFly = function(){ return true; };   
window.isDeadly = function(){ return true; };   
assert( isNimble() && canFly() && isDeadly(), "Still works, even though isNimble is moved." );   
function isNimble(){ return true; }

### Where can assignments be accessed?

assert( typeof canFly == "undefined", "canFly doesn't get that benefit." );   
assert( typeof isDeadly == "undefined", "Nor does isDeadly." );   
var canFly = function(){ return true; };   
window.isDeadly = function(){ return true; };

### Can functions be defined below return statements?

function stealthCheck(){   
  assert( stealth(), "We'll never get below the return, but that's OK!" );   
   
  return stealth();   
   
  function stealth(){ return true; }   
}   
   
stealthCheck();

**Named Function**

### We can refer to a function, within itself, by its name.

function yell(n){   
  return n > 0 ? yell(n-1) + "a" : "hiy";   
}   
assert( yell(4) == "hiyaaaa", "Calling the

### What is the name of a function?

var ninja = function myNinja(){   
  assert( ninja == myNinja, "This function is named two things - at once!" );   
};   
ninja();   
assert( typeof myNinja == "undefined", "But myNinja isn't defined outside of the function." );   
log( ninja );

### We can even do it if we're an anonymous function that's an object property.

var ninja = {   
  yell: function(n){   
    return n > 0 ? ninja.yell(n-1) + "a" : "hiy";   
  }   
};   
assert( ninja.yell(4) == "hiyaaaa", "A single object isn't too bad, either." );

### But what happens when we remove the original object?

var ninja = {   
  yell: function(n){   
    return n > 0 ? ninja.yell(n-1) + "a" : "hiy";   
  }   
};   
assert( ninja.yell(4) == "hiyaaaa", "A single object isn't too bad, either." );   
   
var samurai = { yell: ninja.yell };   
var ninja = null;   
   
try {   
  samurai.yell(4);   
} catch(e){   
  assert( false, "Uh, this isn't good! Where'd ninja.yell go?" );   
}

### Let's give the anonymous function a name!

var ninja = {   
  yell: function yell(n){   
    return n > 0 ? yell(n-1) + "a" : "hiy";   
  }   
};   
assert( ninja.yell(4) == "hiyaaaa", "Works as we would expect it to!" );   
   
var samurai = { yell: ninja.yell };   
var ninja = {};   
assert( samurai.yell(4) == "hiyaaaa", "The method correctly calls itself." );

### What if we don't want to give the function a name?

var ninja = {   
  yell: function(n){   
    return n > 0 ? arguments.callee(n-1) + "a" : "hiy";   
  }   
};   
assert( ninja.yell(4) == "hiyaaaa", "arguments.callee is the function itself." );

### How similar are functions and objects?

var obj = {};   
var fn = function(){};   
assert( obj && fn, "Both the object and function exist." );

### How similar are functions and objects?

var obj = {};   
var fn = function(){};   
obj.prop = "some value";   
fn.prop = "some value";   
assert( obj.prop == fn.prop, "Both are objects, both have the property." );

### Is it possible to cache the return results from a function?

function getElements( name ) {   
  var results;   
   
  if ( getElements.cache[name] ) {   
    results = getElements.cache[name];   
  } else {   
    results = document.getElementsByTagName(name);   
    getElements.cache[name] = results;   
  }   
   
  return results;   
}   
getElements.cache = {};   
   
log( "Elements found: ", getElements("pre").length );   
log( "Cache found: ", getElements.cache.pre.length );

### QUIZ: Can you cache the results of this function?

function isPrime( num ) {   
  var prime = num != 1; // Everything but 1 can be prime   
  for ( var i = 2; i < num; i++ ) {   
    if ( num % i == 0 ) {   
      prime = false;   
      break;   
    }   
  }   
  return prime;   
}   
   
assert( isPrime(5), "Make sure the function works, 5 is prime." );   
assert( isPrime.cache[5], "Is the answer cached?" );

### One possible way to cache the results:

function isPrime( num ) {   
  if ( isPrime.cache[ num ] != null )   
    return isPrime.cache[ num ];   
     
  var prime = num != 1; // Everything but 1 can be prime   
  for ( var i = 2; i < num; i++ ) {   
    if ( num % i == 0 ) {   
      prime = false;   
      break;   
    }   
  }   
    
  isPrime.cache[ num ] = prime   
    
  return prime;   
}   
   
isPrime.cache = {};   
   
assert( isPrime(5), "Make sure the function works, 5 is prime." );   
assert( isPrime.cache[5], "Make sure the answer is cached." );

**Context**

### What happens if a function is an object property?

var katana = {   
  isSharp: true,   
  use: function(){   
    this.isSharp = !this.isSharp;   
  }   
};   
katana.use();   
assert( !katana.isSharp, "Verify the value of isSharp has been changed." );

### What exactly does context represent?

function katana(){   
  this.isSharp = true;   
}   
katana();   
assert( isSharp === true, "A global object now exists with that name and value." );   
   
var shuriken = {   
  toss: function(){   
    this.isSharp = true;   
  }   
};   
shuriken.toss();   
assert( shuriken.isSharp === true, "When it's an object property, the value is set within the object." );

### How can we change the context of a function?

var object = {};   
function fn(){   
  return this;   
}   
assert( fn() == this, "The context is the global object." );   
assert( fn.call(object) == object, "The context is changed to a specific object." );

### Different ways of changing the context:

function add(a, b){   
  return a + b;   
}   
assert( add.call(this, 1, 2) == 3, ".call() takes individual arguments" );   
assert( add.apply(this, [1, 2]) == 3, ".apply() takes an array of arguments" );

### QUIZ: How can we implement looping with a callback?

function loop(array, fn){   
  for ( var i = 0; i < array.length; i++ ) {   
    // Implement me!   
  }   
}   
var num = 0;   
loop([0, 1, 2], function(value){   
  assert(value == num++, "Make sure the contents are as we expect it.");   
  assert(this instanceof Array, "The context should be the full array.");   
});

### A possible solution for function looping:

function loop(array, fn){   
  for ( var i = 0; i < array.length; i++ )   
    fn.call( array, array[i], i );   
}   
var num = 0;   
loop([0, 1, 2], function(value, i){   
  assert(value == num++, "Make sure the contents are as we expect it.");   
  assert(this instanceof Array, "The context should be the full array.");   
});

**Instance**

### What does the new operator do?

function Ninja(){   
  this.name = "Ninja";   
}   
   
var ninjaA = Ninja();   
assert( !ninjaA, "Is undefined, not an instance of Ninja." );   
   
var ninjaB = new Ninja();   
assert( ninjaB.name == "Ninja", "Property exists on the ninja instance." );

### We have a 'this' context that is a Ninja object.

function Ninja(){   
  this.swung = false;   
     
  // Should return true   
  this.swingSword = function(){   
    this.swung = !this.swung;   
    return this.swung;   
  };   
}   
   
var ninja = new Ninja();   
assert( ninja.swingSword(), "Calling the instance method." );   
assert( ninja.swung, "The ninja has swung the sword." );   
   
var ninjaB = new Ninja();   
assert( !ninjaB.swung, "Make sure that the ninja has not swung his sword." );

### QUIZ: Add a method that gives a name to the ninja.

function Ninja(name){   
  // Implement!   
}   
   
var ninja = new Ninja("John");   
assert( ninja.name == "John", "The name has been set on initialization" );   
   
ninja.changeName("Bob");   
assert( ninja.name == "Bob", "The name was successfully changed." );

### Add a new property and method to the object.

function Ninja(name){   
  this.changeName = function(name){   
    this.name = name;   
  };   
   
  this.changeName( name );   
}   
   
var ninja = new Ninja("John");   
assert( ninja.name == "John", "The name has been set on initialization" );   
   
ninja.changeName("Bob");   
assert( ninja.name == "Bob", "The name was successfully changed." );

### What happens when we forget to use the new operator?

function User(first, last){   
  this.name = first + " " + last;   
}   
   
var user = User("John", "Resig");   
assert( typeof user == "undefined", "Since new wasn't used, the instance is undefined." );

### What happens when we forget to use the new operator? (cont.)

function User(first, last){   
  this.name = first + " " + last;   
}   
   
window.name = "Resig";   
var user = User("John", name);   
   
assert( name == "John Resig", "The name variable is accidentally overridden." );

### We need to make sure that the new operator is always used.

function User(first, last){   
  if ( !(this instanceof User) )   
    return new User(first, last);   
     
  this.name = first + " " + last;   
}   
   
var name = "Resig";   
var user = User("John", name);   
   
assert( user, "This was defined correctly, even if it was by mistake." );   
assert( name == "Resig", "The right name was maintained." );

### QUIZ: Is there another, more generic, way of doing this?

function User(first, last){   
  if ( !(this instanceof \_\_\_) )   
    return new User(first, last);   
     
  this.name = first + " " + last;   
}   
   
var name = "Resig";   
var user = User("John", name);   
   
assert( user, "This was defined correctly, even if it was by mistake." );   
assert( name == "Resig", "The right name was maintained." );

### A solution using arguments.callee.

function User(first, last){   
  if ( !(this instanceof arguments.callee) )   
    return new User(first, last);   
     
  this.name = first + " " + last;   
}   
   
var name = "Resig";   
var user = User("John", name);   
   
assert( user, "This was defined correctly, even if it was by mistake." );   
assert( name == "Resig", "The right name was maintained." );

### Using a variable number of arguments to our advantage.

function merge(root){   
  for ( var i = 1; i < arguments.length; i++ )   
    for ( var key in arguments[i] )   
      root[key] = arguments[i][key];   
  return root;   
}   
   
var merged = merge({name: "John"}, {city: "Boston"});   
assert( merged.name == "John", "The original name is intact." );   
assert( merged.city == "Boston", "And the city has been copied over." );

### How can we find the Min/Max number in an array?

function smallest(array){   
  return Math.min.apply( Math, array );   
}   
function largest(array){   
  return Math.max.apply( Math, array );   
}   
assert(smallest([0, 1, 2, 3]) == 0, "Locate the smallest value.");   
assert(largest([0, 1, 2, 3]) == 3, "Locate the largest value.");

### Another possible solution:

function smallest(){   
  return Math.min.apply( Math, arguments );   
}   
function largest(){   
  return Math.max.apply( Math, arguments );   
}   
assert(smallest(0, 1, 2, 3) == 0, "Locate the smallest value.");   
assert(largest(0, 1, 2, 3) == 3, "Locate the largest value.");

### Uh oh, what's going wrong here?

function highest(){   
  return arguments.sort(function(a,b){   
    return b - a;   
  });   
}   
assert(highest(1, 1, 2, 3)[0] == 3, "Get the highest value.");   
assert(highest(3, 1, 2, 3, 4, 5)[1] == 4, "Verify the results.");

### QUIZ: We must convert array-like objects into actual arrays. Can any built-in methods help?

// Hint: Arrays have .slice and .splice methods which return new arrays.   
function highest(){   
  return makeArray(arguments).slice(1).sort(function(a,b){   
    return b - a;   
  });   
}   
   
function makeArray(array){   
  // Implement me!   
}   
   
// Expecting: [3,2,1]   
assert(highest(1, 1, 2, 3)[0] == 3, "Get the highest value.");   
// Expecting: [5,4,3,2,1]   
assert(highest(3, 1, 2, 3, 4, 5)[1] == 4, "Verify the results.");

### We can use built-in methods to our advantage.

function highest(){   
  return makeArray(arguments).sort(function(a,b){   
    return b - a;   
  });   
}   
   
function makeArray(array){   
  return Array().slice.call( array );   
}   
   
assert(highest(1, 1, 2, 3)[0] == 3, "Get the highest value.");   
assert(highest(3, 1, 2, 3, 4, 5)[1] == 4, "Verify the results.");

### QUIZ: Implement a multiplication function (first argument by largest number).

function multiMax(multi){   
  // Make an array of all but the first argument   
  var allButFirst = \_\_\_;   
   
  // Find the largest number in that array of arguments   
  var largestAllButFirst = \_\_\_;   
   
  // Return the multiplied result   
  return multi \* largestAllButFirst;   
}   
assert( multiMax(3, 1, 2, 3) == 9, "3\*3=9 (First arg, by largest.)" );

### We can use call and apply to build a solution.

function multiMax(multi){   
  // Make an array of all but the first argument   
  var allButFirst = Array().slice.call( arguments, 1 );   
   
  // Find the largest number in that array of arguments   
  var largestAllButFirst = Math.max.apply( Math, allButFirst );   
   
  // Return the multiplied result   
  return multi \* largestAllButFirst;   
}   
assert( multiMax(3, 1, 2, 3) == 9, "3\*3=9 (First arg, by largest.)" );

### A basic closure.

var num = 10;   
   
function addNum(myNum){   
  return num + myNum;   
}   
   
assert( addNum(5) == 15, "Add two numbers together, one from a closure." );

### But why doesn't this work?

var num = 10;   
   
function addNum(myNum){   
  return num + myNum;   
}   
   
num = 15;   
   
assert( addNum(5) == 15, "Add two numbers together, one from a closure." );

### Closures are frequently used for callbacks.

var results = jQuery("#results").html("<li>Loading...</li>");   
   
jQuery.get("test.html", function(html){   
  results.html( html );   
  assert( results, "The element to append to, via a closure." );   
});

### They're also useful for timers.

var count = 0;   
   
var timer = setInterval(function(){   
  if ( count < 5 ) {   
    log( "Timer call: ", count );   
    count++;   
  } else {   
    assert( count == 5, "Count came via a closure, accessed each step." );   
    assert( timer, "The timer reference is also via a closure." );   
    clearInterval( timer );   
  }   
}, 100);

### and they're also frequently used when attaching event listeners.

var count = 1;   
var elem = document.createElement("li");   
elem.innerHTML = "Click me!";   
elem.onclick = function(){   
  log( "Click #", count++ );   
};   
document.getElementById("results").appendChild( elem );   
assert( elem.parentNode, "Clickable element appended." );

### QUIZ: What are the values of the variables?

var a = 5;   
function runMe(a){   
 assert( a == \_\_\_, "Check the value of a." );   
   
 function innerRun(){   
   assert( b == \_\_\_, "Check the value of b." );   
   assert( c == \_\_\_, "Check the value of c." );   
 }   
   
 var b = 7;   
 innerRun();   
 var c = 8;   
}   
runMe(6);   
   
for ( var d = 0; d < 3; d++ ) {   
 setTimeout(function(){   
   assert( d == \_\_\_, "Check the value of d." );   
 }, 100);   
}

### How does a function's length property work?

function makeNinja(name){}   
function makeSamurai(name, rank){}   
assert( makeNinja.length == 1, "Only expecting a single argument" );   
assert( makeSamurai.length == 2, "Multiple arguments expected" );

### We can use it to implement method overloading.

function addMethod(object, name, fn){   
  // Save a reference to the old method   
  var old = object[ name ];   
   
  // Overwrite the method with our new one   
  object[ name ] = function(){   
    // Check the number of incoming arguments,   
    // compared to our overloaded function   
    if ( fn.length == arguments.length )   
      // If there was a match, run the function   
      return fn.apply( this, arguments );   
   
    // Otherwise, fallback to the old method   
    else if ( typeof old === "function" )   
      return old.apply( this, arguments );   
  };   
}

### How method overloading might work, using the function length property.

function addMethod(object, name, fn){   
  // Save a reference to the old method   
  var old = object[ name ];   
   
  // Overwrite the method with our new one   
  object[ name ] = function(){   
    // Check the number of incoming arguments,   
    // compared to our overloaded function   
    if ( fn.length == arguments.length )   
      // If there was a match, run the function   
      return fn.apply( this, arguments );   
   
    // Otherwise, fallback to the old method   
    else if ( typeof old === "function" )   
      return old.apply( this, arguments );   
  };   
}   
   
function Ninjas(){   
  var ninjas = [ "Dean Edwards", "Sam Stephenson", "Alex Russell" ];   
  addMethod(this, "find", function(){   
    return ninjas;   
  });   
  addMethod(this, "find", function(name){   
    var ret = [];   
    for ( var i = 0; i < ninjas.length; i++ )   
      if ( ninjas[i].indexOf(name) == 0 )   
        ret.push( ninjas[i] );   
    return ret;   
  });   
  addMethod(this, "find", function(first, last){   
    var ret = [];   
    for ( var i = 0; i < ninjas.length; i++ )   
      if ( ninjas[i] == (first + " " + last) )   
        ret.push( ninjas[i] );   
    return ret;   
  });   
}   
   
var ninjas = new Ninjas();   
assert( ninjas.find().length == 3, "Finds all ninjas" );   
assert( ninjas.find("Sam").length == 1, "Finds ninjas by first name" );   
assert( ninjas.find("Dean", "Edwards").length == 1, "Finds ninjas by first and last name" );   
assert( ninjas.find("Alex", "X", "Russell") == null, "Does nothing" );

# Interfaces

The easiest way to see how interfaces work is to start with a simple example:

**function** **printLabel**(labeledObj: { label: string }) {

console.log(labeledObj.label);

}

**let** myObj = {size: 10, label: "Size 10 Object"};

printLabel(myObj);

The type checker checks the call to printLabel. The printLabel function has a single parameter that requires that the object passed in has a property called label of type string. Notice that our object actually has more properties than this, but the compiler only checks that at least the ones required are present and match the types required. There are some cases where TypeScript isn’t as lenient, which we’ll cover in a bit.

We can write the same example again, this time using an interface to describe the requirement of having the labelproperty that is a string:

**interface** LabeledValue {

label: string;

}

**function** **printLabel**(labeledObj: LabeledValue) {

console.log(labeledObj.label);

}

**let** myObj = {size: 10, label: "Size 10 Object"};

printLabel(myObj);

The interface labeledValue is a name we can now use to describe the requirement in the previous example. It still represents having a single property called label that is of type string. Notice we didn’t have to explicitly say that the object we pass to printLabel implements this interface like we might have to in other languages. Here, it’s only the shape that matters. If the object we pass to the function meets the requirements listed, then it’s allowed.

It’s worth pointing out that the type checker does not require that these properties come in any sort of order, only that the properties the interface requires are present and have the required type.

# Optional Properties

Not all properties of an interface may be required. Some exist under certain conditions or may not be there at all. These optional properties are popular when creating patterns like “option bags” where you pass an object to a function that only has a couple of properties filled in.

Here’s an example of this pattern:

**interface** SquareConfig {

color?: string;

width?: number;

}

**function** **createSquare**(config: SquareConfig): {color: string; area: number} {

**let** newSquare = {color: "white", area: 100};

**if** (config.color) {

newSquare.color = config.color;

}

**if** (config.width) {

newSquare.area = config.width \* config.width;

}

**return** newSquare;

}

**let** mySquare = createSquare({color: "black"});

Interfaces with optional properties are written similar to other interfaces, with each optional property denoted by a ? at the end of the property name in the declaration.

The advantage of optional properties is that you can describe these possibly available properties while still also preventing use of properties that are not part of the interface. For example, had we mistyped the name of the colorproperty in createSquare, we would get an error message letting us know:

**interface** SquareConfig {

color?: string;

width?: number;

}

**function** **createSquare**(config: SquareConfig): { color: string; area: number } {

**let** newSquare = {color: "white", area: 100};

**if** (config.clor) {

// Error: Property 'clor' does not exist on type 'SquareConfig'

newSquare.color = config.clor;

}

**if** (config.width) {

newSquare.area = config.width \* config.width;

}

**return** newSquare;

}

**let** mySquare = createSquare({color: "black"});

# Readonly properties

Some properties should only be modifiable when an object is first created. You can specify this by putting readonlybefore the name of the property:

**interface** Point {

readonly x: number;

readonly y: number;

}

You can construct a Point by assigning an object literal. After the assignment, x and y can’t be changed.

**let** p1: Point = { x: 10, y: 20 };

p1.x = 5; // error!

TypeScript comes with a ReadonlyArray<T> type that is the same as Array<T> with all mutating methods removed, so you can make sure you don’t change your arrays after creation:

**let** a: number[] = [1, 2, 3, 4];

**let** ro: ReadonlyArray<number> = a;

ro[0] = 12; // error!

ro.push(5); // error!

ro.length = 100; // error!

a = ro; // error!

On the last line of the snippet you can see that even assigning the entire ReadonlyArray back to a normal array is illegal. You can still override it with a type assertion, though:

a = ro as number[];

## readonly vs const

The easiest way to remember whether to use readonly or const is to ask whether you’re using it on a variable or a property. Variables use const whereas properties use readonly.

# Excess Property Checks

In our first example using interfaces, TypeScript lets us pass { size: number; label: string; } to something that only expected a { label: string; }. We also just learned about optional properties, and how they’re useful when describing so-called “option bags”.

However, combining the two naively would allow an error to sneak in. For example, taking our last example using createSquare:

**interface** SquareConfig {

color?: string;

width?: number;

}

**function** **createSquare**(config: SquareConfig): { color: string; area: number } {

// ...

}

**let** mySquare = createSquare({ colour: "red", width: 100 });

Notice the given argument to createSquare is spelled *colour* instead of color. In plain JavaScript, this sort of thing fails silently.

You could argue that this program is correctly typed, since the width properties are compatible, there’s no colorproperty present, and the extra colour property is insignificant.

However, TypeScript takes the stance that there’s probably a bug in this code. Object literals get special treatment and undergo excess property checking when assigning them to other variables, or passing them as arguments. If an object literal has any properties that the “target type” doesn’t have, you’ll get an error:

// error: 'colour' not expected in type 'SquareConfig'

**let** mySquare = createSquare({ colour: "red", width: 100 });

Getting around these checks is actually really simple. The easiest method is to just use a type assertion:

**let** mySquare = createSquare({ width: 100, opacity: 0.5 } as SquareConfig);

However, a better approach might be to add a string index signature if you’re sure that the object can have some extra properties that are used in some special way. If SquareConfig can have color and width properties with the above types, but could also have any number of other properties, then we could define it like so:

**interface** SquareConfig {

color?: string;

width?: number;

[propName: string]: any;

}

We’ll discuss index signatures in a bit, but here we’re saying a SquareConfig can have any number of properties, and as long as they aren’t color or width, their types don’t matter.

One final way to get around these checks, which might be a bit surprising, is to assign the object to another variable: Since squareOptions won’t undergo excess property checks, the compiler won’t give you an error.

**let** squareOptions = { colour: "red", width: 100 };

**let** mySquare = createSquare(squareOptions);

The above workaround will work as long as you have a common property between squareOptions and SquareConfig. In this example, it was the property width. It will however, fail if the variable does not have any common object property. For example:

**let** squareOptions = { colour: "red" };

**let** mySquare = createSquare(squareOptions);

Keep in mind that for simple code like above, you probably shouldn’t be trying to “get around” these checks. For more complex object literals that have methods and hold state, you might need to keep these techniques in mind, but a majority of excess property errors are actually bugs. That means if you’re running into excess property checking problems for something like option bags, you might need to revise some of your type declarations. In this instance, if it’s okay to pass an object with both a color or colour property to createSquare, you should fix up the definition of SquareConfig to reflect that.

# Function Types

Interfaces are capable of describing the wide range of shapes that JavaScript objects can take. In addition to describing an object with properties, interfaces are also capable of describing function types.

To describe a function type with an interface, we give the interface a call signature. This is like a function declaration with only the parameter list and return type given. Each parameter in the parameter list requires both name and type.

**interface** SearchFunc {

(source: string, subString: string): boolean;

}

Once defined, we can use this function type interface like we would other interfaces. Here, we show how you can create a variable of a function type and assign it a function value of the same type.

**let** mySearch: SearchFunc;

mySearch = **function**(source: string, subString: string) {

**let** result = source.search(subString);

**return** result > -1;

}

For function types to correctly type check, the names of the parameters do not need to match. We could have, for example, written the above example like this:

**let** mySearch: SearchFunc;

mySearch = **function**(src: string, sub: string): **boolean** {

**let** result = src.search(sub);

**return** result > -1;

}

Function parameters are checked one at a time, with the type in each corresponding parameter position checked against each other. If you do not want to specify types at all, TypeScript’s contextual typing can infer the argument types since the function value is assigned directly to a variable of type SearchFunc. Here, also, the return type of our function expression is implied by the values it returns (here false and true). Had the function expression returned numbers or strings, the type checker would have warned us that return type doesn’t match the return type described in the SearchFunc interface.

**let** mySearch: SearchFunc;

mySearch = **function**(src, sub) {

**let** result = src.search(sub);

**return** result > -1;

}

# Indexable Types

Similarly to how we can use interfaces to describe function types, we can also describe types that we can “index into” like a[10], or ageMap["daniel"]. Indexable types have an index signature that describes the types we can use to index into the object, along with the corresponding return types when indexing. Let’s take an example:

**interface** StringArray {

[index: number]: string;

}

**let** myArray: StringArray;

myArray = ["Bob", "Fred"];

**let** myStr: string = myArray[0];

Above, we have a StringArray interface that has an index signature. This index signature states that when a StringArray is indexed with a number, it will return a string.

There are two types of supported index signatures: string and number. It is possible to support both types of indexers, but the type returned from a numeric indexer must be a subtype of the type returned from the string indexer. This is because when indexing with a number, JavaScript will actually convert that to a string before indexing into an object. That means that indexing with 100 (a number) is the same thing as indexing with "100" (a string), so the two need to be consistent.

**class** Animal {

name: string;

}

**class** Dog extends Animal {

breed: string;

}

// Error: indexing with a numeric string might get you a completely separate type of Animal!

**interface** NotOkay {

[x: number]: Animal;

[x: string]: Dog;

}

While string index signatures are a powerful way to describe the “dictionary” pattern, they also enforce that all properties match their return type. This is because a string index declares that obj.property is also available as obj["property"]. In the following example, name’s type does not match the string index’s type, and the type checker gives an error:

**interface** NumberDictionary {

[index: string]: number;

length: number; // ok, length is a number

name: string; // error, the type of 'name' is not a subtype of the indexer

}

Finally, you can make index signatures readonly in order to prevent assignment to their indices:

**interface** ReadonlyStringArray {

readonly [index: number]: string;

}

**let** myArray: ReadonlyStringArray = ["Alice", "Bob"];

myArray[2] = "Mallory"; // error!

You can’t set myArray[2] because the index signature is readonly.

# Class Types

## Implementing an interface

One of the most common uses of interfaces in languages like C# and Java, that of explicitly enforcing that a class meets a particular contract, is also possible in TypeScript.

**interface** ClockInterface {

currentTime: Date;

}

**class** Clock **implements** ClockInterface {

currentTime: Date = **new** Date();

**constructor**(h: number, m: number) { }

}

You can also describe methods in an interface that are implemented in the class, as we do with setTime in the below example:

**interface** ClockInterface {

currentTime: Date;

setTime(d: Date): void;

}

**class** Clock **implements** ClockInterface {

currentTime: Date = **new** Date();

setTime(d: Date) {

**this**.currentTime = d;

}

**constructor**(h: number, m: number) { }

}

Interfaces describe the public side of the class, rather than both the public and private side. This prohibits you from using them to check that a class also has particular types for the private side of the class instance.

## Difference between the static and instance sides of classes

When working with classes and interfaces, it helps to keep in mind that a class has two types: the type of the static side and the type of the instance side. You may notice that if you create an interface with a construct signature and try to create a class that implements this interface you get an error:

**interface** ClockConstructor {

**new** (hour: number, minute: number);

}

**class** Clock **implements** ClockConstructor {

currentTime: Date;

**constructor**(h: number, m: number) { }

}

This is because when a class implements an interface, only the instance side of the class is checked. Since the constructor sits in the static side, it is not included in this check.

Instead, you would need to work with the static side of the class directly. In this example, we define two interfaces, ClockConstructor for the constructor and ClockInterface for the instance methods. Then, for convenience, we define a constructor function createClock that creates instances of the type that is passed to it:

**interface** ClockConstructor {

**new** (hour: number, minute: number): ClockInterface;

}

**interface** ClockInterface {

tick(): void;

}

**function** **createClock**(ctor: ClockConstructor, hour: number, minute: number): **ClockInterface** {

**return** **new** ctor(hour, minute);

}

**class** DigitalClock **implements** ClockInterface {

**constructor**(h: number, m: number) { }

tick() {

console.log("beep beep");

}

}

**class** AnalogClock **implements** ClockInterface {

**constructor**(h: number, m: number) { }

tick() {

console.log("tick tock");

}

}

**let** digital = createClock(DigitalClock, 12, 17);

**let** analog = createClock(AnalogClock, 7, 32);

Because createClock’s first parameter is of type ClockConstructor, in createClock(AnalogClock, 7, 32), it checks that AnalogClock has the correct constructor signature.

Another simple way is to use class expressions:

**interface** ClockConstructor {

**new** (hour: number, minute: number);

}

**interface** ClockInterface {

tick();

}

**const** Clock: ClockConstructor = **class** Clock **implements** ClockInterface {

**constructor**(h: number, m: number) {}

tick() {

console.log("beep beep");

}

}

# Extending Interfaces

Like classes, interfaces can extend each other. This allows you to copy the members of one interface into another, which gives you more flexibility in how you separate your interfaces into reusable components.

**interface** Shape {

color: string;

}

**interface** Square extends Shape {

sideLength: number;

}

**let** square = <**Square**>{};

square.color = "blue";

square.sideLength = 10;

An interface can extend multiple interfaces, creating a combination of all of the interfaces.

**interface** Shape {

color: string;

}

**interface** PenStroke {

penWidth: number;

}

**interface** Square extends Shape, PenStroke {

sideLength: number;

}

**let** square = <**Square**>{};

square.color = "blue";

square.sideLength = 10;

square.penWidth = 5.0;

# Hybrid Types

As we mentioned earlier, interfaces can describe the rich types present in real world JavaScript. Because of JavaScript’s dynamic and flexible nature, you may occasionally encounter an object that works as a combination of some of the types described above.

One such example is an object that acts as both a function and an object, with additional properties:

**interface** Counter {

(start: number): string;

interval: number;

reset(): void;

}

**function** **getCounter**(): **Counter** {

**let** counter = <**Counter**>function (start: number) { };

counter.interval = 123;

counter.reset = function () { };

return counter;

}

let c = getCounter();

c(10);

c.reset();

c.interval = 5.0;

When interacting with 3rd-party JavaScript, you may need to use patterns like the above to fully describe the shape of the type.

# Interfaces Extending Classes

When an interface type extends a class type it inherits the members of the class but not their implementations. It is as if the interface had declared all of the members of the class without providing an implementation. Interfaces inherit even the private and protected members of a base class. This means that when you create an interface that extends a class with private or protected members, that interface type can only be implemented by that class or a subclass of it.

This is useful when you have a large inheritance hierarchy, but want to specify that your code works with only subclasses that have certain properties. The subclasses don’t have to be related besides inheriting from the base class. For example:

**class** Control {

**private** state: any;

}

**interface** SelectableControl extends Control {

select(): void;

}

**class** Button extends Control **implements** SelectableControl {

select() { }

}

**class** TextBox extends Control {

select() { }

}

// Error: Property 'state' is missing in type 'Image'.

**class** Image **implements** SelectableControl {

**private** state: any;

select() { }

}

**class** Location {

}

In the above example, SelectableControl contains all of the members of Control, including the private stateproperty. Since state is a private member it is only possible for descendants of Control to implement SelectableControl. This is because only descendants of Control will have a state private member that originates in the same declaration, which is a requirement for private members to be compatible.

Within the Control class it is possible to access the state private member through an instance of SelectableControl. Effectively, a SelectableControl acts like a Control that is known to have a select method. The Button and TextBox classes are subtypes of SelectableControl (because they both inherit from Control and have a select method), but the Image and Location classes are not.

# Enums

Enums allow us to define a set of named constants. Using enums can make it easier to document intent, or create a set of distinct cases. TypeScript provides both numeric and string-based enums. (Read only connection, which will be constant)

## Numeric enums

We’ll first start off with numeric enums, which are probably more familiar if you’re coming from other languages. An enum can be defined using the enum keyword.

**enum** Direction {

Up = 1,

Down,

Left,

Right,

}

Direction.Left

Above, we have a numeric enum where Up is initialized with 1. All of the following members are auto-incremented from that point on. In other words, Direction.Up has the value 1, Down has 2, Left has 3, and Right has 4.

If we wanted, we could leave off the initializers entirely:

**enum** Direction {

Up, --0

Down, --1

Left, --2

Right, --3

} – default index will start from 0

Direction.Up

Here, Up would have the value 0, Down would have 1, etc. This auto-incrementing behavior is useful for cases where we might not care about the member values themselves, but do care that each value is distinct from other values in the same enum.

Using an enum is simple: just access any member as a property off of the enum itself, and declare types using the name of the enum:

**enum** Response {

No = 0,

Yes = 1,

}

**function** **respond**(recipient: string, message: Response): **void** {

// ...

}

respond("Princess Caroline", Response.Yes)

Numeric enums can be mixed in computed and constant members (see below). The short story is, enums without initializers either need to be first, or have to come after numeric enums initialized with numeric constants or other constant enum members. In other words, the following isn’t allowed:

**enum** E {

A = getSomeValue(),

B, // error! 'A' is not constant-initialized, so 'B' needs an initializer

}

## String enums

String enums are a similar concept, but have some subtle runtime differences as documented below. In a string enum, each member has to be constant-initialized with a string literal, or with another string enum member.

**enum** Direction {

Up = "UP",

Down = "DOWN",

Left = "LEFT",

Right = "RIGHT",

}

Read the enum:

Direction.Up

While string enums don’t have auto-incrementing behavior, string enums have the benefit that they “serialize” well. In other words, if you were debugging and had to read the runtime value of a numeric enum, the value is often opaque - it doesn’t convey any useful meaning on its own (though reverse mapping can often help), string enums allow you to give a meaningful and readable value when your code runs, independent of the name of the enum member itself.

## Heterogeneous enums (Mix type)

## Heterogeneous (one type : either numeric, or string)

Technically enums can be mixed with string and numeric members, but it’s not clear why you would ever want to do so:

**enum** BooleanLikeHeterogeneousEnum {

No = 0,

Yes = "YES",

}

Unless you’re really trying to take advantage of JavaScript’s runtime behavior in a clever way, it’s advised that you don’t do this.

## Computed and constant members

Each enum member has a value associated with it which can be either constant or computed. An enum member is considered constant if:

* It is the first member in the enum and it has no initializer, in which case it’s assigned the value 0:
* // E.X is constant:
* **enum** E { X }
* It does not have an initializer and the preceding enum member was a numeric constant. In this case the value of the current enum member will be the value of the preceding enum member plus one.
* // All enum members in 'E1' and 'E2' are constant.
* **enum** E1 { X, Y, Z }
* **enum** E2 {
* A = 1, B, C

}

**Ref:** https://javascript.info