

TypeScript 3.2 is now available. [Download \(/#download-links\)](#) our latest version today!

Documentation ▼

Functions

Introduction

Functions are the fundamental building block of any applications in JavaScript. They're how you build up layers of abstraction, mimicking classes, information hiding, and modules. In TypeScript, while there are classes, namespaces, and modules, functions still play the key role in describing how to *do* things. TypeScript also adds some new capabilities to the standard JavaScript functions to make them easier to work with.

Functions

To begin, just as in JavaScript, TypeScript functions can be created both as a named function or as an anonymous function. This allows you to choose the most appropriate approach for your application, whether you're building a list of functions in an API or a one-off function to hand off to another function.

To quickly recap what these two approaches look like in JavaScript:

```
// Named function
function add(x, y) {
    return x + y;
}

// Anonymous function
let myAdd = function(x, y) { return x + y; };
```

Just as in JavaScript, functions can refer to variables outside of the function body. When they do so, they're said to *capture* these variables. While understanding how this works, and the trade-offs when using this technique, are outside of the scope of this article, having a firm understanding how this mechanic is an important piece of working with JavaScript and TypeScript.

```
let z = 100;

function addToZ(x, y) {
    return x + y + z;
}
```

Function Types

Typing the function

Let's add types to our simple examples from earlier:

```
function add(x: number, y: number): number {  
    return x + y;  
}  
  
let myAdd = function(x: number, y: number): number { return x + y; };
```

We can add types to each of the parameters and then to the function itself to add a return type. TypeScript can figure the return type out by looking at the return statements, so we can also optionally leave this off in many cases.

Writing the function type

Now that we've typed the function, let's write the full type of the function out by looking at each piece of the function type.

```
let myAdd: (x: number, y: number) => number =  
    function(x: number, y: number): number { return x + y; };
```

A function's type has the same two parts: the type of the arguments and the return type. When writing out the whole function type, both parts are required. We write out the parameter types just like a parameter list, giving each parameter a name and a type. This name is just to help with readability. We could have instead written:

```
let myAdd: (baseValue: number, increment: number) => number =  
    function(x: number, y: number): number { return x + y; };
```

As long as the parameter types line up, it's considered a valid type for the function, regardless of the names you give the parameters in the function type.

The second part is the return type. We make it clear which is the return type by using a fat arrow (`=>`) between the parameters and the return type. As mentioned before, this is a required part of the function type, so if the function doesn't return a value, you would use `void` instead of leaving it off.

Of note, only the parameters and the return type make up the function type. Captured variables are not reflected in the type. In effect, captured variables are part of the "hidden state" of any function and do not make up its API.

Inferring the types

In playing with the example, you may notice that the TypeScript compiler can figure out the type if you have types on one side of the equation but not the other:

```
// myAdd has the full function type  
let myAdd = function(x: number, y: number): number { return x + y; };  
  
// The parameters 'x' and 'y' have the type number  
let myAdd: (baseValue: number, increment: number) => number =  
    function(x, y) { return x + y; };
```

This is called "contextual typing", a form of type inference. This helps cut down on the amount of effort to keep your program typed.

Optional and Default Parameters

In TypeScript, every parameter is assumed to be required by the function. This doesn't mean that it can't be given `null` or `undefined`, but rather, when the function is called the compiler will check that the user has provided a value for each parameter. The compiler also assumes that these parameters are the only parameters that will be passed to the function. In short, the number of arguments given to a function has to match the number of parameters the function expects.

```
function buildName(firstName: string, lastName: string) {  
    return firstName + " " + lastName;  
}  
  
let result1 = buildName("Bob");           // error, too few parameters  
let result2 = buildName("Bob", "Adams", "Sr."); // error, too many parameters  
let result3 = buildName("Bob", "Adams");    // ah, just right
```

In JavaScript, every parameter is optional, and users may leave them off as they see fit. When they do, their value is `undefined`. We can get this functionality in TypeScript by adding a `?` to the end of parameters we want to be optional. For example, let's say we want the last name parameter from above to be optional:

```
function buildName(firstName: string, lastName?: string) {  
    if (lastName)  
        return firstName + " " + lastName;  
    else  
        return firstName;  
}  
  
let result1 = buildName("Bob");           // works correctly now  
let result2 = buildName("Bob", "Adams", "Sr."); // error, too many parameters  
let result3 = buildName("Bob", "Adams");    // ah, just right
```

Any optional parameters must follow required parameters. Had we wanted to make the first name optional rather than the last name, we would need to change the order of parameters in the function, putting the first name last in the list.

In TypeScript, we can also set a value that a parameter will be assigned if the user does not provide one, or if the user passes `undefined` in its place. These are called default-initialized parameters. Let's take the previous example and default the last name to "Smith".

```
function buildName(firstName: string, lastName = "Smith") {  
    return firstName + " " + lastName;  
}  
  
let result1 = buildName("Bob");           // works correctly now, returns "Bob Smith"  
let result2 = buildName("Bob", undefined); // still works, also returns "Bob Smith"  
let result3 = buildName("Bob", "Adams", "Sr."); // error, too many parameters  
let result4 = buildName("Bob", "Adams");    // ah, just right
```

Default-initialized parameters that come after all required parameters are treated as optional, and just like optional parameters, can be omitted when calling their respective function. This means optional parameters and trailing default parameters will share commonality in their types, so both

```
function buildName(firstName: string, lastName?: string) {  
    // ...  
}
```

and

```
function buildName(firstName: string, lastName = "Smith") {  
    // ...  
}
```

share the same type (`firstName: string, lastName?: string`) => `string`. The default value of `lastName` disappears in the type, only leaving behind the fact that the parameter is optional.

Unlike plain optional parameters, default-initialized parameters don't *need* to occur after required parameters. If a default-initialized parameter comes before a required parameter, users need to explicitly pass `undefined` to get the default initialized value. For example, we could write our last example with only a default initializer on `firstName`:

```
function buildName(firstName = "Will", lastName: string) {
    return firstName + " " + lastName;
}

let result1 = buildName("Bob");           // error, too few parameters
let result2 = buildName("Bob", "Adams", "Sr."); // error, too many parameters
let result3 = buildName("Bob", "Adams");    // okay and returns "Bob Adams"
let result4 = buildName(undefined, "Adams"); // okay and returns "Will Adams"
```

Rest Parameters

Required, optional, and default parameters all have one thing in common: they talk about one parameter at a time. Sometimes, you want to work with multiple parameters as a group, or you may not know how many parameters a function will ultimately take. In JavaScript, you can work with the arguments directly using the `arguments` variable that is visible inside every function body.

In TypeScript, you can gather these arguments together into a variable:

```
function buildName(firstName: string, ...restOfName: string[]) {
    return firstName + " " + restOfName.join(" ");
}

let employeeName = buildName("Joseph", "Samuel", "Lucas", "MacKinzie");
```

Rest parameters are treated as a boundless number of optional parameters. When passing arguments for a rest parameter, you can use as many as you want; you can even pass none. The compiler will build an array of the arguments passed in with the name given after the ellipsis (`...`), allowing you to use it in your function.

The ellipsis is also used in the type of the function with rest parameters:

```
function buildName(firstName: string, ...restOfName: string[]) {
    return firstName + " " + restOfName.join(" ");
}

let buildNameFun: (fname: string, ...rest: string[]) => string = buildName;
```

this

Learning how to use `this` in JavaScript is something of a rite of passage. Since TypeScript is a superset of JavaScript, TypeScript developers also need to learn how to use `this` and how to spot when it's not being used correctly. Fortunately, TypeScript lets you catch incorrect uses of `this` with a couple of techniques. If you need to learn how `this` works in JavaScript, though, first read Yehuda Katz's *Understanding JavaScript Function Invocation and "this"* (<http://yehudakatz.com/2011/08/11/understanding-javascript-function-invocation-and-this/>). Yehuda's article explains the inner workings of `this` very well, so we'll just cover the basics here.

this and arrow functions

In JavaScript, `this` is a variable that's set when a function is called. This makes it a very powerful and flexible feature, but it comes at the cost of always having to know about the context that a function is executing in. This is notoriously confusing, especially when returning a function or passing a function as an argument.

Let's look at an example:

```
let deck = {
  suits: ["hearts", "spades", "clubs", "diamonds"],
  cards: Array(52),
  createCardPicker: function() {
    return function() {
      let pickedCard = Math.floor(Math.random() * 52);
      let pickedSuit = Math.floor(pickedCard / 13);

      return {suit: this.suits[pickedSuit], card: pickedCard % 13};
    }
  }
}

let cardPicker = deck.createCardPicker();
let pickedCard = cardPicker();

alert("card: " + pickedCard.card + " of " + pickedCard.suit);
```

Notice that `createCardPicker` is a function that itself returns a function. If we tried to run the example, we would get an error instead of the expected alert box. This is because the `this` being used in the function created by `createCardPicker` will be set to `window` instead of our `deck` object. That's because we call `cardPicker()` on its own. A top-level non-method syntax call like this will use `window` for `this`. (Note: under strict mode, `this` will be `undefined` rather than `window`).

We can fix this by making sure the function is bound to the correct `this` before we return the function to be used later. This way, regardless of how it's later used, it will still be able to see the original `deck` object. To do this, we change the function expression to use the ECMAScript 6 arrow syntax. Arrow functions capture the `this` where the function is created rather than where it is invoked:

```
let deck = {
  suits: ["hearts", "spades", "clubs", "diamonds"],
  cards: Array(52),
  createCardPicker: function() {
    // NOTE: the line below is now an arrow function, allowing us to capture 'this' right here
    return () => {
      let pickedCard = Math.floor(Math.random() * 52);
      let pickedSuit = Math.floor(pickedCard / 13);

      return {suit: this.suits[pickedSuit], card: pickedCard % 13};
    }
  }
}

let cardPicker = deck.createCardPicker();
let pickedCard = cardPicker();

alert("card: " + pickedCard.card + " of " + pickedCard.suit);
```

Even better, TypeScript will warn you when you make this mistake if you pass the `--noImplicitThis` flag to the compiler. It will point out that `this` in `this.suits[pickedSuit]` is of type `any`.

this parameters

Unfortunately, the type of `this.suits[pickedSuit]` is still `any`. That's because `this` comes from the function expression inside the object literal. To fix this, you can provide an explicit `this` parameter. `this` parameters are fake parameters that come first in the parameter list of a function:

```
function f(this: void) {
    // make sure `this` is unusable in this standalone function
}
```

Let's add a couple of interfaces to our example above, `Card` and `Deck`, to make the types clearer and easier to reuse:

```
interface Card {
    suit: string;
    card: number;
}
interface Deck {
    suits: string[];
    cards: number[];
    createCardPicker(this: Deck): () => Card;
}
let deck: Deck = {
    suits: ["hearts", "spades", "clubs", "diamonds"],
    cards: Array(52),
    // NOTE: The function now explicitly specifies that its callee must be of type Deck
    createCardPicker: function(this: Deck) {
        return () => {
            let pickedCard = Math.floor(Math.random() * 52);
            let pickedSuit = Math.floor(pickedCard / 13);

            return {suit: this.suits[pickedSuit], card: pickedCard % 13};
        }
    }
}

let cardPicker = deck.createCardPicker();
let pickedCard = cardPicker();

alert("card: " + pickedCard.card + " of " + pickedCard.suit);
```

Now TypeScript knows that `createCardPicker` expects to be called on a `Deck` object. That means that `this` is of type `Deck` now, not `any`, so `--noImplicitThis` will not cause any errors.

this parameters in callbacks

You can also run into errors with `this` in callbacks, when you pass functions to a library that will later call them. Because the library that calls your callback will call it like a normal function, `this` will be `undefined`. With some work you can use `this` parameters to prevent errors with callbacks too. First, the library author needs to annotate the callback type with `this`:

```
interface UIElement {
    addClickListener(onclick: (this: void, e: Event) => void): void;
}
```

`this: void` means that `addClickListener` expects `onclick` to be a function that does not require a `this` type. Second, annotate your calling code with `this`:

```
class Handler {
  info: string;
  onClickBad(this: Handler, e: Event) {
    // oops, used this here. using this callback would crash at runtime
    this.info = e.message;
  }
}
let h = new Handler();
uiElement.addEventListener(h.onClickBad); // error!
```

With `this` annotated, you make it explicit that `onClickBad` must be called on an instance of `Handler`. Then TypeScript will detect that `addEventListener` requires a function that has `this: void`. To fix the error, change the type of `this`:

```
class Handler {
  info: string;
  onClickGood(this: void, e: Event) {
    // can't use this here because it's of type void!
    console.log('clicked!');
  }
}
let h = new Handler();
uiElement.addEventListener(h.onClickGood);
```

Because `onClickGood` specifies its `this` type as `void`, it is legal to pass to `addEventListener`. Of course, this also means that it can't use `this.info`. If you want both then you'll have to use an arrow function:

```
class Handler {
  info: string;
  onClickGood = (e: Event) => { this.info = e.message }
}
```

This works because arrow functions don't capture `this`, so you can always pass them to something that expects `this: void`. The downside is that one arrow function is created per object of type `Handler`. Methods, on the other hand, are only created once and attached to `Handler`'s prototype. They are shared between all objects of type `Handler`.

Overloads

JavaScript is inherently a very dynamic language. It's not uncommon for a single JavaScript function to return different types of objects based on the shape of the arguments passed in.

```

let suits = ["hearts", "spades", "clubs", "diamonds"];

function pickCard(x): any {
  // Check to see if we're working with an object/array
  // if so, they gave us the deck and we'll pick the card
  if (typeof x == "object") {
    let pickedCard = Math.floor(Math.random() * x.length);
    return pickedCard;
  }
  // Otherwise just let them pick the card
  else if (typeof x == "number") {
    let pickedSuit = Math.floor(x / 13);
    return { suit: suits[pickedSuit], card: x % 13 };
  }
}

let myDeck = [{ suit: "diamonds", card: 2 }, { suit: "spades", card: 10 }, { suit: "hearts", card: 4 }];
let pickedCard1 = myDeck[pickCard(myDeck)];
alert("card: " + pickedCard1.card + " of " + pickedCard1.suit);

let pickedCard2 = pickCard(15);
alert("card: " + pickedCard2.card + " of " + pickedCard2.suit);

```

Here the `pickCard` function will return two different things based on what the user has passed in. If the user passes in an object that represents the deck, the function will pick the card. If the user picks the card, we tell them which card they've picked. But how do we describe this to the type system?

The answer is to supply multiple function types for the same function as a list of overloads. This list is what the compiler will use to resolve function calls. Let's create a list of overloads that describe what our `pickCard` accepts and what it returns.

```

let suits = ["hearts", "spades", "clubs", "diamonds"];

function pickCard(x: {suit: string; card: number; }[]): number;
function pickCard(x: number): {suit: string; card: number; };
function pickCard(x): any {
  // Check to see if we're working with an object/array
  // if so, they gave us the deck and we'll pick the card
  if (typeof x == "object") {
    let pickedCard = Math.floor(Math.random() * x.length);
    return pickedCard;
  }
  // Otherwise just let them pick the card
  else if (typeof x == "number") {
    let pickedSuit = Math.floor(x / 13);
    return { suit: suits[pickedSuit], card: x % 13 };
  }
}

let myDeck = [{ suit: "diamonds", card: 2 }, { suit: "spades", card: 10 }, { suit: "hearts", card: 4 }];
let pickedCard1 = myDeck[pickCard(myDeck)];
alert("card: " + pickedCard1.card + " of " + pickedCard1.suit);

let pickedCard2 = pickCard(15);
alert("card: " + pickedCard2.card + " of " + pickedCard2.suit);


```

With this change, the overloads now give us type-checked calls to the `pickCard` function.

In order for the compiler to pick the correct typecheck, it follows a similar process to the underlying JavaScript. It looks at the overload list, and proceeding with the first overload attempts to call the function with the provided parameters. If it finds a match, it picks this overload as the correct overload. For this reason, it's customary to order overloads from most specific to least specific.

Note that the function `pickCard(x): any` piece is not part of the overload list, so it only has two overloads: one that takes an object and one that takes a number. Calling `pickCard` with any other parameter types would cause an error.

Made with ❤ in Redmond Follow @Typescriptlang (<https://twitter.com/typescriptlang>)

Privacy (<https://go.microsoft.com/fwlink/?LinkId=521839>) ©2012-2018 Microsoft  Microsoft