TypeScript

JavaScript that scales.

TypeScript is a typed superset of JavaScript that compiles to plain JavaScript.
Any browser,. Any host. Any OS. Open source.

TypeScript

- Functions
- Fat arrow
- Type Inference
- Type Compatibility
- Namespaces

Functions

```
// Named function
```

```
// Anonymous function
```

Functions

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

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

Functions

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

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

Functions

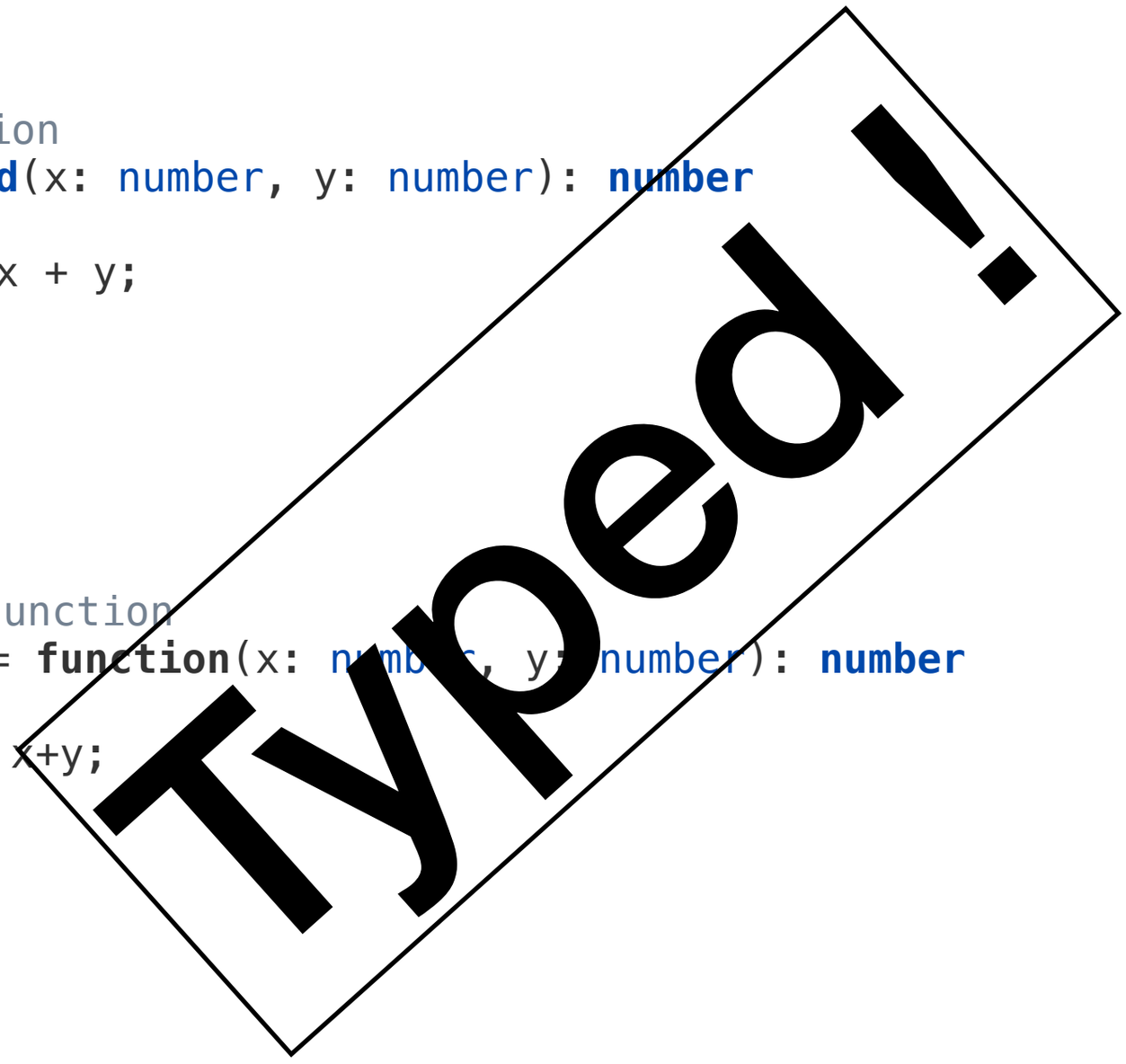
```
// Named function
    function add(x: number, y: number): number
    {
        return x + y;
    }
```

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

Functions

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

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



Functions

```
// Default Parameters
```

Functions

```
// Default Parameters
function buildName(firstName: string, lastName: string) {
  if (lastName)
    return firstName + " " + lastName;
  else
    return firstName;
}

let result1 = buildName("Bob");
let result2 = buildName("Bob", "Adams", "Sr.");
let result3 = buildName("Bob", "Adams");
```

// every parameter is assumed to be required by the function.

Functions

```
// Default Parameters
function buildName(firstName: string, lastName: string) {
  if (lastName)
    return firstName + " " + lastName;
  else
    return firstName;
}

let result1 = buildName("Bob");                  // error, missing parameters
let result2 = buildName("Bob", "Adams", "Sr.");  // error, too many parameters
let result3 = buildName("Bob", "Adams");         // ah, just right
```

```
// every parameter is assumed to be required by the function.
```

Functions

```
// Optional Parameters
```

Functions

```
// Optional Parameters
function buildName(firstName: string, lastName: string) {
  if (lastName)
    return firstName + " " + lastName;
  else
    return firstName;
}

let result1 = buildName("Bob");
let result2 = buildName("Bob", "Adams", "Sr.");
let result3 = buildName("Bob", "Adams");
```

Functions

```
// Optional Parameters
function buildName(firstName: string, lastName?: string) {
  if (lastName)
    return firstName + " " + lastName;
  else
    return firstName;
}

let result1 = buildName("Bob");
let result2 = buildName("Bob", "Adams", "Sr.");
let result3 = buildName("Bob", "Adams");
```

Functions

```
// Optional Parameters
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
```

Functions

```
// Default-Initialised Parameters
```

Functions

```
// Default-Initialised Parameters
```

```
// a value that a parameter will be assigned if the user does not provide one,
// or if the user passes undefined in its place.
```

Functions

```
// Default-Initialised Parameters
function buildName(firstName: string, lastName: string) {
    return firstName + " " + lastName;
}
```

```
// a value that a parameter will be assigned if the user does not provide one,
// or if the user passes undefined in its place.
```

Functions

```
// Default-Initialised Parameters
function buildName(firstName = "Will", lastName: string) {
    return firstName + " " + lastName;
}
```

```
// a value that a parameter will be assigned if the user does not provide one,
// or if the user passes undefined in its place.
```

Functions

```
// Default-Initialised Parameters
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"
```

```
// a value that a parameter will be assigned if the user does not provide one,
// or if the user passes undefined in its place.
```

Functions

```
// Default-Initialised Parameters
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"
```

```
// a value that a parameter will be assigned if the user does not provide one,
// or if the user passes undefined in its place.
```

Functions

```
// Rest parameters
```

Functions

```
// Rest parameters
// allow you to quickly accept multiple arguments in your function
// and get them as an array.
```

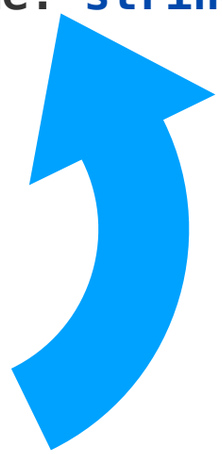
Functions

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

// allow you to quickly accept multiple arguments in your function
// and get them as an array.

Functions

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



```
...restOfName: string[]
```

```
// allow you to quickly accept multiple arguments in your function
// and get them as an array.
```

Functions

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

let employeeName = buildName(“Joseph”, “Samuel”, “Lucas”,”Mackenzie”);
```

```
// allow you to quickly accept multiple arguments in your function
// and get them as an array.
```

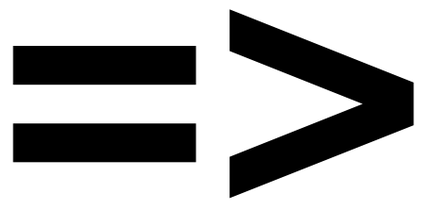
Fat Arrow

=>

Fat Arrow

The motivation for a *fat arrow* is:

1. You don't need to keep typing function
2. It lexically captures the meaning of this
3. It lexically captures the meaning of arguments

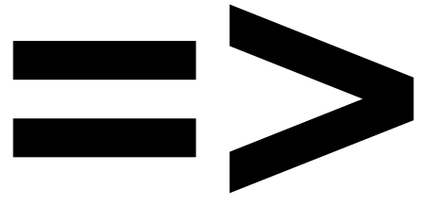


Fat Arrow

The motivation for a *fat arrow* is:

1. You don't need to keep typing function
2. It lexically captures the meaning of this
3. It lexically captures the meaning of arguments

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

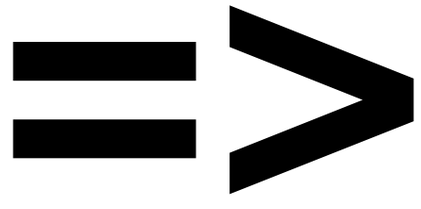


Fat Arrow

The motivation for a *fat arrow* is:

1. You don't need to keep typing function
2. It lexically captures the meaning of this
3. It lexically captures the meaning of arguments

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

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



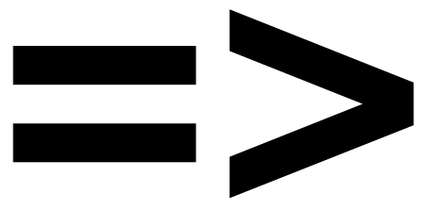
Fat Arrow

```
// Example
function Person(age) {
    this.age = age;
    this.growOld = function() {
        this.age++;
    }
}
var person = new Person(1);
setTimeout(person.growOld,1000);

setTimeout(function() { console.log(person.age); },2000);
```

```
=>
```

Fat Arrow

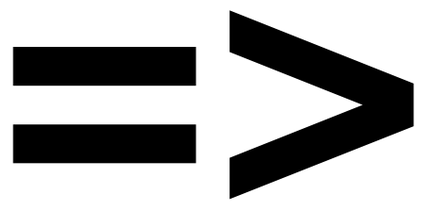
```
// Example
function Person(age) {
    this.age = age;
    this.growOld = function() {
        this.age++;
    }
}
var person = new Person(1);
setTimeout(person.growOld,1000);

setTimeout(function() { console.log(person.age); },2000); // 1, should have
been 2
```



Fat Arrow

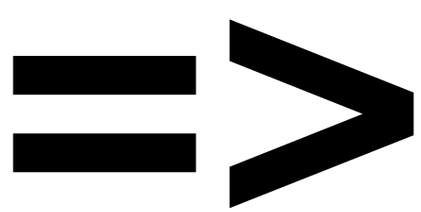
```
// Example
function Person(age) {
    this.age = age;
    this.growOld = () => {
        this.age++;
    }
}
var person = new Person(1);
setTimeout(person.growOld,1000);

setTimeout(function() { console.log(person.age); },2000); // 2
```

```
// captures the meaning of this
// from the surrounding context.
```



Fat Arrow

```
// Refactor example to combine arrows with classes
class Person {
    constructor(public age:number) {}
    growOld = () => {
        this.age++;
    }
}
var person = new Person(1);
setTimeout(person.growOld,1000);

setTimeout(function() { console.log(person.age); },2000); // 2
```



Type Inference

Type Inference

```
// provides type information
// when there is no explicit type annotation.
```

Type Inference

```
let x = 3;
```

// provides type information
// when there is no explicit type annotation.

Type Inference

```
let x = 3;

let x = [0, 1, null];
```

```
// provides type information
// when there is no explicit type annotation.
```

Type Inference

```
let x = 3;

let x = [0, 1, null];

let zoo = [new Rhino(), new Elephant(), new Snake()];
```

```
// provides type information
// when there is no explicit type annotation.
```

Type Inference

```
let x = 3;

let x = [0, 1, null];

let zoo: Animal[] = [new Rhino(), new Elephant(), new Snake()];
```

```
// provides type information
// when there is no explicit type annotation.
```

Type Compatibility

```
// x is compatible with y if y has at
   least the same members as x.
```

Type Compatibility

```
interface Thing {
    name: string;
}

class Person {
    name: string;
}
```

```
let p: Thing;

p = new Person();
```

Type Compatibility

```
interface Thing {
    name: string;
}

class Person {
    name: string;
}
```

```
let p: Thing;
// OK, because of structural typing
p = new Person();
```

Type Compatibility

```
interface Thing {
    name: string;
}

let x: Thing;

let y = { name: "Alice", location: "Seattle" };
x = y;
```

Type Compatibility

```
interface Thing {
    name: string;
}

let x: Thing;
// y's inferred type is { name: string; location: string; }
let y = { name: "Alice", location: "Seattle" };
x = y;
```

Type Compatibility

```
interface Thing {
    name: string;
}

let x: Thing;
// y's inferred type is { name: string; location: string; }
let y = { name: "Alice", location: "Seattle" };
x = y;
```

```
// x is compatible with y if y has at
  least the same members as x.
```

Type Compatibility

```
// Function compatibility
```

```
// Each parameter in x must have a
   corresponding parameter in y with a
   compatible type.
```

Type Compatibility

```
// Function compatibility
let x = (a: number) => 0;
let y = (b: number, s: string) => 0;

y = x; // OK
x = y; // Error
```

```
// Each parameter in x must have a
  corresponding parameter in y with a
  compatible type.
```

Namespaces

```
// Namespaces are simply named JavaScript objects
// in the global namespace.
```

Namespaces

```
namespace Utility {
    export function log(msg) {
        console.log(msg);
    }
    export function error(msg) {
        console.error(msg);
    }
}
```

```
// usage
Utility.log('Call me');
Utility.error('maybe!');
```

// Namespaces are simply named JavaScript objects
// in the global namespace.

Exercices

- Typescript quality (typed, functions, modules, interface)
- Solved problems
- Team work (or not)
- 15/20 minutes **work** - 10 mins **discussion**