Electronics and
Computer Science

UNIVERSITY OF
Southampton

Working with Functions in TypeScript

COMP3207/COMP6244

Dr A. Rezazadeh

October 18

Function Declarations and Function Expressions

- In the previous lecture, we introduced the possibility of **declaring** functions with (**named function**) or without (**unnamed** or **anonymous** function) explicitly indicating its name, but we didn't mention that we were also using two different types of function.
- In the following example, the named function `greetNamed` is a **function declaration** while `greetUnnamed` is a **function expression**.

Function Declarations vs Function Expressions

- Ignore the first two lines, which contain two console log statements, for now:

```
console.log(greetNamed("John"));
console.log(greetUnnamed("John"));

function greetNamed(name : string) : string {
    if(name) {
      return "Hi! " + name;
    }
}


var greetUnnamed = function(name : string) : string {
 if(name){


     return "Hi! " + name;
   }

 }
```

Function Declarations vs Function Expressions

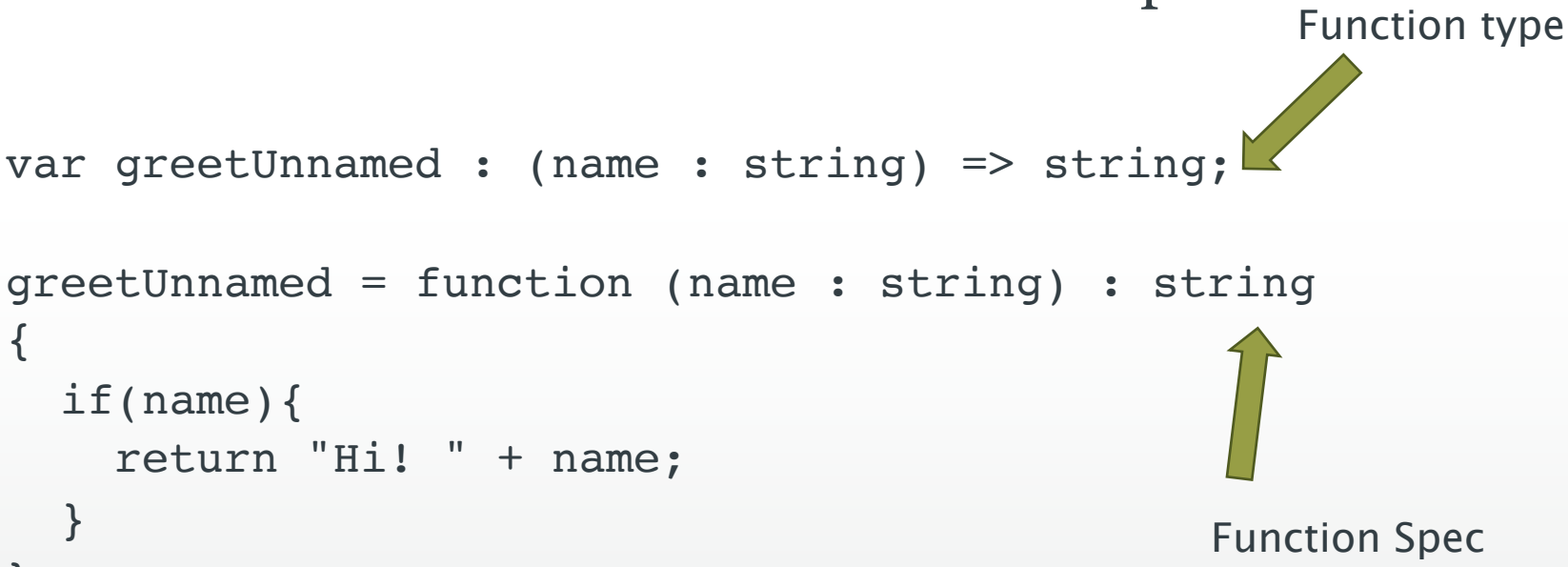
- We might think that these preceding functions are really similar, but they will **behave differently**.
- The interpreter can evaluate a **function declaration** as it is being parsed.
- On the other hand, the **function expression** is part of an assignment and will **not be evaluated** until the assignment has been completed.

Function Declarations vs Function Expressions

- If we compile the preceding TypeScript code snippet into JavaScript and try to execute it in a web browser, we will observe that the first alert statement will work because JavaScript knows about the declaration function and can parse it before the program is executed.
- However, the second alert statement will throw an exception, which indicates that `greetUnnamed` is not a function. The exception is thrown because the `greetUnnamed` assignment must be completed before the function can be evaluated.

Function type & Function Specification

- In Function **function expressions** we will need to not just specify the types of the function elements, but also the function itself. Let's take a look at an example:



Combing Function Type & Declaration

- We will assign a function to this variable in the same line in which it is declared. The assigned function must be equal to the variable type.

```
var greetUnnamed : (name : string) => string = function(name : string)
: string {
  if(name){
    return "Hi! " + name;
  }
}
```

Functions with Optional Parameters

- Unlike JavaScript, the TypeScript compiler will throw an error if we attempt to invoke a function without providing the exact number and type of parameters that its signature declares. Lets look at this function:

```
function add(foo : number, bar : number, foobar : number) : number {
  return foo + bar + foobar;
}
```

- Attempting to invoke this function without providing exactly three numbers, we will get a compilation error :

```
add();          // Supplied parameters do not match any signature
add(2, 2);      // Supplied parameters do not match any signature
add(2, 2, 2); // returns 6
```

Functions with Optional Parameters – Cont.

- There are scenarios in which we might want to be able to call the function without providing all its arguments.
- TypeScript features optional parameters in functions to help us to increase the flexibility of our functions.

```
function add(foo : number, bar : number, foobar? : number) : number {
  var result =  foo + bar;
  if(foobar !== undefined){
    result += foobar;
  }
  return result;
}
```

- Note how we have changed the `foobar` parameter name into **`foobar?`**, and how we are checking the type of `foobar` inside the function to identify if the parameter was supplied as an argument to the function or not.

Functions with Optional Parameters – Cont.

```
function add(foo : number, bar : number, foobar? : number) : number {
  var result =  foo + bar;
  if(foobar !== undefined){
    result += foobar;
  }
  return result;
}
```

It is important to note that the optional parameters must always be located after the required parameters in the function's parameters list.

- After doing these changes, the TypeScript compiler will allow us to invoke the function without errors when we supply two or three arguments to it:

```
add();          // Supplied parameters do not match any signature
add(2, 2);      // returns 4
add(2, 2, 2); // returns 6
```

Functions with Default Parameters

- There are some scenarios in which it would be more useful to provide a default value for a parameter when it is not supplied.

```
function add(foo : number, bar : number, foobar? : number) :
number {
  return foo + bar + (foobar !== undefined ? foobar : 0);
}
```

- There is nothing wrong with the preceding function, but we can improve its readability by providing a default value for the foobar parameter instead of flagging it as an optional parameter:

```
function add(foo : number, bar : number, foobar : number = 0) :
number {
  return foo + bar + foobar;
}
```

Just like optional parameters, default parameters must be always located after any required parameters in the function's parameter list.

Functions with Default Parameters

- To indicate that a function parameter is optional, we just need to provide a default value using the = operator when declaring the function's signature.
- The TypeScript compiler will generate an if structure in the JavaScript output to set a default value for the foobar parameter if it is not passed as an argument to the function:

```
function add(foo, bar, foobar) {
    if (foobar === void 0) { foobar = 0; }
    return foo + bar + foobar;
}
```

`Void 0` issued by the TypeScript compiler to check if a variable is equal to undefined. While most developers use the undefined variable, most compilers use `void 0`.

Functions with **rest** Parameters

- We have seen how to use optional parameters to increase the number of ways that we can invoke a function.
- But what if we wanted to allow other developers to pass four or five parameters to our function?
- And what if we wanted to allow them to pass as many parameters as they may need?
- The solution to this possible scenario is the use of **rest** parameters.
- The **rest** parameter syntax allows us to represent an **indefinite** number of arguments as an array:

Functions with **rest** Parameters

```
function add(...foo : number[]) : number {
  var result = 0;
  for(var i = 0; i < foo.length; i++){
    result += foo[i];
  }
  return result;
}
```

Note that the name of the parameter foo is preceded by an ellipsis (a set of three periods—not the actual ellipsis character)

We can now invoke the add function with as many parameters as we may need:

```
add();                  // returns 0
add(2);                 // returns 2
add(2,2);               // returns 4
add(2,2,2);             // returns 6
add(2,2,2,2);           // returns 8
add(2,2,2,2,2);         // returns 10
add(2,2,2,2,2,2);       // returns 12
```

Functions with **rest** Parameters

- JavaScript functions have a built-in object called the `arguments` object.
- This object is available as a local variable named arguments. The `arguments` variable contains an object similar to an array, which contains the arguments used when the function was invoked.
- If we examine the JavaScript output for the previous code snippet, we will notice that TypeScript iterates the arguments object in order to add the values to the foo variable:

JavaScript Output

```
function add() {
    var foo = [];
    for (var _i = 0; _i < arguments.length; _i++) {
        foo[_i - 0] = arguments[_i];
    }
    var result = 0;
    for (var i = 0; i < foo.length; i++) {
        result += foo[i];
    }
    return result;
}
```

- We can argue that this is an extra, unnecessary iteration over the function's parameters.

Avoiding the rest Parameters

- You may want to consider avoiding the use of rest parameters and use an array as the only parameter of the function instead:

```
function add(foo : number[]) : number {
  var result = 0;
  for(var i = 0; i < foo.length; i++){
    result += foo[i];
  }
  return result;
}
```

- This function takes an array of numbers as its only parameter. The invocation API will be a little different from the rest parameters, but we will effectively avoid the extra iteration over the function's argument list

The Invocation of the above Function

```
add();       // Supplied parameters do not match any signature
add(2);      // Supplied parameters do not match any signature
add(2,2);    // Supplied parameters do not match any signature
add(2,2,2); // Supplied parameters do not match any signature

add([]);       // returns 0
add([2]);      // returns 2
add([2,2]);    // returns 4
add([2,2,2]); // returns 6
```

Function Overloading

- Function **overloading** or **method overloading** is the ability to create **multiple methods** with the **same name** and a **different number** of **parameters** or **types**.
- In TypeScript, we can overload a function by specifying all function signatures of a function, followed by a signature known as the **implementation signature**.

Function Overloading

- Let's take a look at an example:

```
function test(name: string) : string;     // overloaded signature
function test(age: number) : string;      // overloaded signature
function test(single: boolean) : string; // overloaded signature
function test(value: (string | number | boolean)) : string; {  //
implementation signature
  switch(typeof value){
    case "string":
      return `My name is ${value}.`;
    case "number":
      return `I'm ${value} years old.`;
    case "boolean":
      return value ? "I'm single." : "I'm not single.";
    default:
      console.log("Invalid Operation!");
  }
}
```

Template Strings & Placeholders

- You might not be familiar with the syntax used in some of the strings in the preceding code snippet. This syntax is known as **Template Strings**.
- Template strings are enclosed by the back-tick (` `) character instead of double or single quotes.
- Template strings can contain ***placeholders***. These are indicated by the **dollar sign and curly braces** (`${expression}`).
- The expressions in the placeholders and the text between them get passed to a function. The default function just concatenates the parts into a single string.

Invoking the Overloaded Function

- The implementation signature must be compatible with all the overloaded signatures, always be the last in the list, and take any or a union type as the type of its parameters.

```
test("Remo");                    // returns "My name is Remo."
test(26);                        // returns "I'm 26 years old.";
test(false);                     // returns "I'm not single.";
test({ custom : "custom" });     // error
```

- The implementation signature can **not** be invocate directly and will cause a compilation error.

Specialised Overloading Signatures

- We can use a **specialised** signature to create **multiple methods** with the **same name** and **number of parameters** but a **different return type**.
- To create a specialized signature, we must indicate the type of function parameter using a string.
- The string literal is used to identify which of the function overloads is invoked:

```
interface Document {
  createElement(tagName: "div"): HTMLDivElement; // specialized
  createElement(tagName: "span"): HTMLSpanElement;  // specialized
  createElement(tagName: "canvas"): HTMLCanvasElement; //
  specialized
  createElement(tagName: string): HTMLElement; // non-specialized
}
```

Specialised Overloading Signatures

- When we declare a specialised signature in an object, it must be assignable to at least one non-specialised signature in the same object.
- This can be observed in the preceding example, as the `createElement` property belongs to a type that contains three specialised signatures, all of which are assignable to the non-specialised signature in the type.
- When writing overloaded declarations, we must list the non-specialised signature last.
- Remember that, as seen in *the previous lecture*, we can also use **union types** to create a method with the same name and number of parameters but a different type.

Function Scope

- While in many programming languages, variables are scoped to a block (a section of code delimited by curly braces {}), in TypeScript (and JavaScript), **variables** are **scoped** to a **function**:

```
function foo() : void {
 if(true){
  var bar : number = 0;
 }
 alert(bar);
}

foo(); // shows 0
```

We might think that the preceding code sample would throw an error in the fifth line because the `bar` variable should be out of the scope when the alert function is invoked. However, if we invoke the `foo` function, the alert function will be able to display the variable bar without errors because all the variables inside a function will be in the scope of the entire function body, even if they are inside another block of code (except a function block).

Variables Hoisting

- This might seem really confusing, but it is easy to understand once we know that, at runtime, all the variable declarations are moved to the top of a function before the function is executed.
- This behaviour is called **hoisting**.

```
function foo() : void {
  var bar :number;
  if(true){
    bar= 0;
  }
  alert(bar);
}
```

- So, before the preceding code snippet is executed, the runtime will move the declaration of the variable bar to the top of our function.
-

Using a Variable Before Declaring it

- This means that we can use a variable before it is declared. Let's take a look at an example:

```
function foo2() : void {
  bar = 0;
  var bar : number;
  alert(bar);
}

foo2();
```

- Because developers with a Java or C# background are not used to the function scope, it is one of the most criticized characteristics of JavaScript.
- The people in charge of the development of the ECMAScript 6 specification are aware of this and, as a result, they have introduced the keywords `let` and `const`.

Block Scope with `let` Keyword

- The `let` keyword allows us to set the scope of a variable to a block(`if, while, for`...) rather than a function block.
- We can update the first example in this section to showcase how `let` works:

```
function foo() : void {
  if(true){
let bar : number = 0;
bar = 1;
  }
  alert(bar); // error
}
```

Variables defined with `const` Keyword

- While variables defined with `const` follow the same scope rules as variables declared with let, they can't be reassigned:

```
function foo() : void {
 if(true){
  const bar : number = 0;
  bar = 1; // error
 }
 alert(bar); // error
}
```

Immediately invoked functions

- An **immediately invoked function expression** (**IIFE**) is a design pattern that produces a lexical scope using function scoping.
- IIFE can be used to avoid variable hoisting from within blocks or to prevent us from polluting the global scope.
- For example:

```
var bar = 0; // global

(function() {
  var foo : number = 0; // in scope of this function
  bar = 1; // in global scope
  console.log(bar); // 1
  console.log(foo); // 0
})();

console.log(bar); // 1
console.log(foo); // error
```

Explaining the Above Example

- In the preceding example, we have wrapped the declaration of a variable (`foo`) with an IIFE.
  - The foo variable is scoped to the IIFE function and is not available in the global scope, which explains the error that is thrown when we try to access it on the last line.
- The `bar` variable is a global. Therefore, it can be accessed from both the inside and the outside of the IIFE function.

Passing Variables to IIFE

- We can also pass a variable to the IIFE to have better control over the creation of variables outside its own scope:

```
1   (function IIFE(msg, times) {
2       for (var i = 1; i <= times; i++) {
3           console.log(msg);
4       }
5   }("Hello!", 5));
```

  - In the above example, on line 1, IIFE has two formal parameters named `msg`, `times` respectively.
  - When we execute the IIFE on line 5, instead of the empty parentheses `()` we have seen so far, we are now passing arguments to the IIFE.
  - Lines 2 and 3 use those parameters inside the IIFE.

Electronics and
Computer Science

UNIVERSITY OF
Southampton

Asynchronous Programming in TypeScript

Asynchronous Programming in TypeScript

- When you execute a task ***synchronously***, you wait for it to finish before moving on to the next line of code.
- When you execute a task ***asynchronously***, the program moves to the next line of code before the task finishes.
- Think of synchronous programming like **waiting in line** and asynchronous programming like **taking a ticket**.
  - When you take a ticket you can go do other things and then be notified when ready.
  - One way to program asynchronously is to use ***callbacks***.

Callbacks and Higher-order Functions

- In JavaScript, functions are **objects**. Because of this, functions can take **functions as arguments**, and can be **returned by other functions**.
  - The function passed to another as an argument is known as a **callback**.
  - *A callback is a function that is to be executed **after** another function has finished executing—hence the name 'call back'.*
  - The functions that accept functions as parameters (callbacks) or return functions as an argument are known as **higher-order functions**.
  - Callbacks are usually anonymous functions.

Why do we need Callbacks?

- For one very important reason—JavaScript is an event driven language.
- This means that instead of **waiting** for a **response** before moving on, JavaScript will keep executing while listening for other events.
- Lets look at a basic example:

```
function first(){
  console.log(1);
}

function second(){
  console.log(2);
}

first();
second();
```

As you would expect, the function first is executed first, and the function second is executed second—logging the following to the console:

```
// 1
// 2
```

Callbacks

- But what if function `first` contains some sort of code that can't be executed immediately?
- For example, an API request where we have to send the request then wait for a response?
- To simulate this action, were going to use `setTimeout` which is a JavaScript function that calls a function after a set amount of time.
- We'll delay our function for 500 milliseconds to simulate an API request. Our new code will look like this:

Callbacks – Cont.

- It's not important that you understand how `setTimeout()` works right now. All that matters is that you see we've moved our `console.log(1);` inside of our 500 millisecond delay. So what happens now when we invoke our functions?

```
function first(){
  // Simulate a code delay
  setTimeout( function(){
    console.log(1);
  }, 500 );
}

function second(){
  console.log(2);
}

first();
second();
```

```
first();
second();

// 2
// 1
```

Even though we invoked the `first()` function first, we logged out the result of that function after the `second()` function.

Callback – Cont.

- It's not that JavaScript didn't execute our functions in the order we wanted it to, it's instead that **JavaScript didn't wait for a response from** `first()` **before moving on to execute** `second()`.
- So why show you this? Because you can't just call one function after another and hope they execute in the right order.
- Callbacks are a way to make sure certain code doesn't execute until other code has already finished execution.

Creating a CallBack

- We created the function `doHomework`:

```
function doHomework(subject) {
  alert(`Starting my ${subject} homework.`);
}
```

- Our function takes one variable, the subject that we are working on.
- Call your function by typing the following into your console:

```
doHomework('math');

// Alerts: Starting my math homework.
```

Creating a CallBack – Cont.

- Now lets add in our callback—as our last parameter in the `doHomework()` function we can pass in callback.
- The callback function is then defined in the second argument of our call to `doHomework()`.

```
function doHomework(subject, callback) {
  alert(`Starting my ${subject} homework.`);
  callback();
}

doHomework('math', function() {
  alert('Finished my homework');
});
```

if you type the above code into your console you will get two alerts back to back: Your 'starting homework' alert, followed by your 'finished homework' alert.

Creating a CallBack – Cont.

- But callback functions don't always have to be defined in our function call. They can be defined elsewhere in our code like this:

```
function doHomework(subject, callback) {
  alert(`Starting my ${subject} homework.`);
  callback();
}

function alertFinished(){
  alert('Finished my homework');
}

doHomework('math', alertFinished);
```

This result of this example is exactly the same as the previous example, but the setup is a little different. As you can see, we've passed the `alertFinished` function definition as an argument during our `doHomework()` function call!

Callbacks – Another Example

```
function doAsyncTask(cb) {
  setTimeout(() => {
    console.log("Async Task Calling Callback");
    cb();
  }, 1000);
}

doAsyncTask(() => console.log("Callback Called"));
```

- We pass to an asynchronous function a function which it will call when the task is completed.
- The `doAsyncTask` function when called kicks of an asynchronous task and ***returns immediately***.
- To **get notified** when the **async task completes** we pass to `doAsyncTask` a function which it will call when the task completes.
- **`cb`** is a short hand form the *callback* function.

Promises

- The use of callbacks can lead to some maintainability problems.
- In ES6 we have an alternative mechanism built into the language called a ***promise***.
- It serves the same function as callbacks but has a nicer syntax and makes it easier to handle errors.
- It can be used to write better **asynchronous** code.
- The core idea behind promises is that a promise represents the result of an **asynchronous operation**.

Promise

- Promise must be in one of the three following states:
  - **Pending**: The initial state of a promise
  - **Fulfilled**: The state of a promise representing a successful operation
  - **Rejected**: The state of a promise representing a failed operation

Promise

- Imagine that you're a top **singer**, and **fans** ask day and night for your **upcoming single**.
- To get some **relief**, you **promise** to **send** it to them **when it's published**.
- You give your fans a **list** to which they can **subscribe** for updates.
  - They can fill in their email addresses, so that when the song becomes available, all **subscribed parties** instantly receive it.
  - And even if **something goes very wrong**, say, if plans to publish the song are cancelled, they will still be **notified**.
- Everyone is happy: you, because the people don't crowd you any more, and fans, because they won't miss the single.

Promise – Cont.

- The above story is a real-life analogy for things we often have in programming:
  1. A "**producing code**" that does something and takes time. For instance, the code loads a remote script. That's a "**singer**".
  2. A "**consuming code**" that wants the result of the "producing code" once it's ready. Many functions may need that result. These are the "**fans**".
  3. A ***promise*** is a **special JavaScript object** that links the "**producing code**" and the "**consuming code**" together.
     - In terms of our analogy: this is the "**subscription list**". The "producing code" takes whatever time it needs to produce the promised result, and the "promise" makes that result available to all of the subscribed code when it's ready.

Defining a new Promise

- The analogy isn't terribly accurate, because JavaScript promises are more complex than a simple subscription list:
  - they have additional features and limitations.
  - But it's fine to begin with.
- The constructor syntax for a promise object is:

```
1 let promise = new Promise(function(resolve, reject) {
2   // executor (the producing code, "singer")
3 });
```

- The function **passed to** new Promise is called the ***executor***. When the promise is created, this executor function runs **automatically**.
- It contains the **producing** code, that should eventually produce a result. In terms of the analogy above: the executor is the "**singer**".

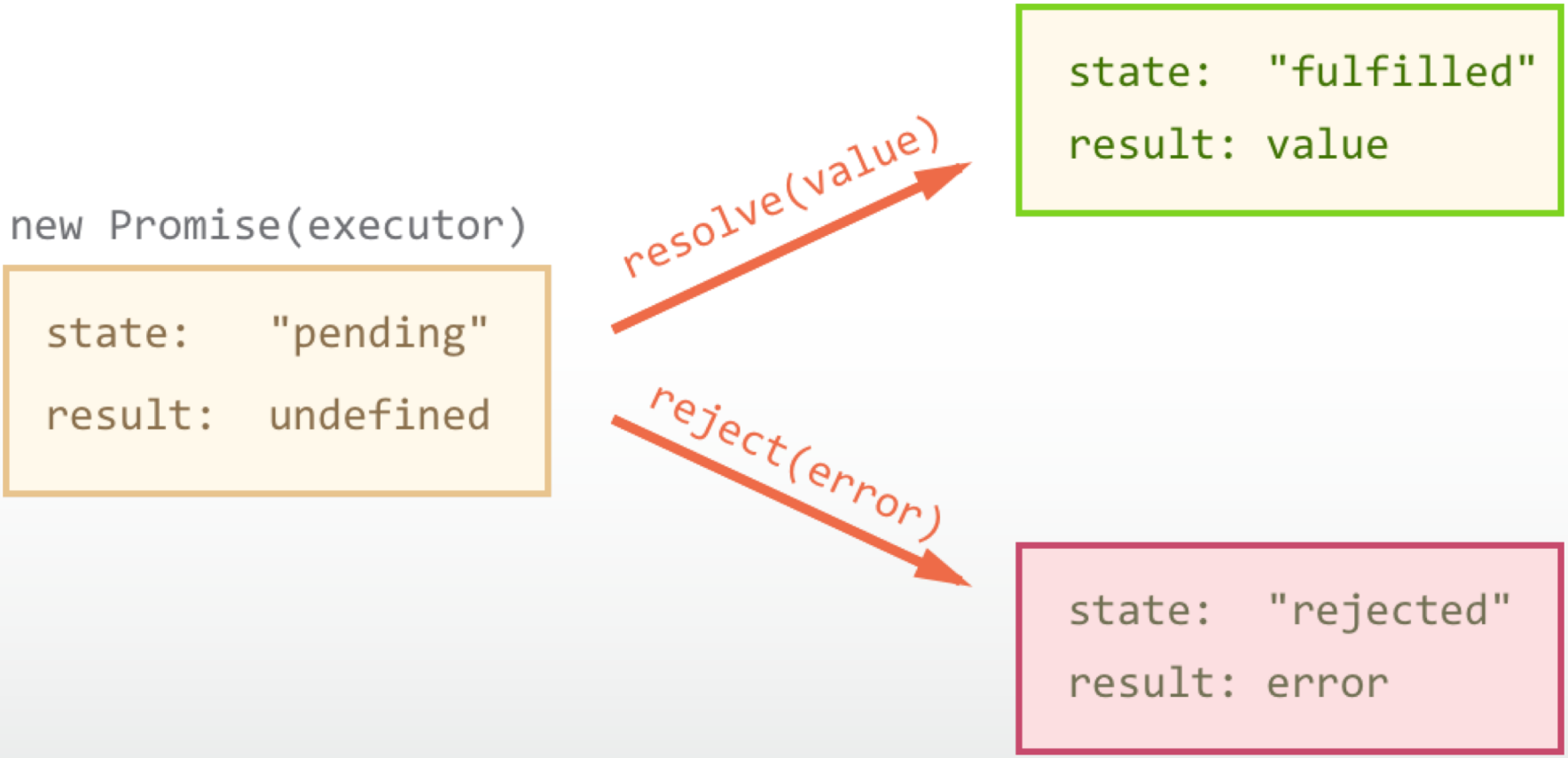
Defining a new Promise – Cont.

- The resulting promise object has internal properties:
  - `state` — initially "**pending**", then changes to either "**fulfilled**" or "**rejected**",
  - `result` — an arbitrary value of your choice, initially "**undefined**".

Defining a new Promise – Cont.

- When the executor finishes the job, it should call one of the functions that it gets as arguments:
  - `resolve(value)` — to indicate that the job finished successfully:
    - sets `state` to "`fulfilled`",
    - sets `result` to `value`.
  - `reject(error)` — to indicate that an error occurred:
    - sets `state` to "`rejected`",
    - sets `result` to `error`.
- Later we'll see how these changes become known to "fans".

Promise – Cont.



An Example of a Promise

- Here's an example of a Promise constructor and a simple executor function with its "**producing code**" (the `setTimeout`):

```
let promise = new Promise(function(resolve, reject) {
  // the function is executed automatically when the promise is constructed

  // after 1 second signal that the job is done with the result "done!"
  setTimeout(() => resolve("done!"), 1000);
});
```

Explaining the Above Example

- We can see two things by running the code above:
  1. The executor is called automatically and immediately (by the new Promise).
  2. The executor receives two arguments: `resolve` and `reject` — these functions are pre-defined by the JavaScript engine. So we don't need to create them. Instead, we should write the executor to call them when ready.
- After one second of "processing" the executor calls `resolve("done")` to produce the result

Another Example of Promise (failed)

- And now an example of the executor **rejecting** the promise with an error:

```
let promise = new Promise(function(resolve, reject) {
  // after 1 second signal that the job is finished with an error
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});
```

- To summarise, the executor should do a job (something that takes time usually) and then call resolve or reject to change the state of the corresponding Promise object.
- The Promise that is either **resolved** or **rejected** is called "**settled**", as opposed to a "**pending**" Promise.

Promise - Some Hints

- The executor should call only one `resolve` or `reject`. The promise's state change is final.
- All further calls of `resolve` and `reject` are ignored:

```
1 let promise = new Promise(function(resolve, reject) {
2   resolve("done");
3
4   reject(new Error("…")); // ignored
5   setTimeout(() => resolve("…")); // ignored
6 });
```

Promise - `Reject` with `Error` objects

- `Reject` with `Error` objects
  - In case if something goes wrong, we can call `reject` with any type of argument (just like `resolve`).
  - But it is recommended to use `Error` objects (or objects that inherit from `Error`).
  - The reasoning for that will soon become apparent.

Promise - Immediately calling `resolve`/`reject`

- In practice, an executor usually does something asynchronously and calls **`resolve`**/`reject` after some time, but it doesn't have to. We also can call `resolve` or `reject` immediately, like this:

```
1 let promise = new Promise(function(resolve, reject) {
2   // not taking our time to do the job
3   resolve(123); // immediately give the result: 123
4 });
```

- For instance, this might happen when we start to do a job but then see that everything has already been completed.
- That's fine. We immediately have a `resolved` Promise, nothing wrong with that.

Promise – The `state` and `result` are internal

- The properties `state` and `result` of the Promise object are internal.
- We can't directly access them from our "consuming code".
- We can use the methods `.then/.catch` for that. They are described below.

Consumers: "`then`" and "`catch`"

- A Promise object serves as a link between the executor (the "**producing code**" or "**singer**) and the consuming functions (the "**fans**"), which will receive the **result** or **error**.
- Consuming functions can be registered (subscribed) using the methods `.then` and `.catch`.
- The syntax of `.then` is:

```
promise.then(
  function(result) { /* handle a successful result */ },
  function(error) { /* handle an error */ }
);
```

The `.then` Function

- The first argument of `.then` is a function that:
  1. runs when the Promise is resolved, and
  2. receives the result.
- The second argument of `.then` is a function that:
  1. runs when the Promise is `rejected`, and
  2. receives the error.

The `.then` Function – Cont.

- For instance, here's the reaction to a successfully resolved promise:

```
1 let promise = new Promise(function(resolve, reject) {
2   setTimeout(() => resolve("done!"), 1000);
3 });
4
5 // resolve runs the first function in .then
6 promise.then(
7   result => alert(result), // shows "done!" after 1 second
8   error => alert(error) // doesn't run
9 );
```

The first function was executed.

The `.then` Function – Cont.

- And in the case of a rejection – the second one:

```
let promise = new Promise(function(resolve, reject) {
  setTimeout(() => reject(new Error("Whoops!")), 1000);
});

// reject runs the second function in .then
promise.then(
  result => alert(result), // doesn't run
  error => alert(error) // shows "Error: Whoops!" after 1 second
);
```

The `.then` Function – Cont.

- If we're interested only in successful completions, then we can provide only one function argument to `.then`:

```
1 let promise = new Promise(resolve => {
2   setTimeout(() => resolve("done!"), 1000);
3 });
4
5 promise.then(alert); // shows "done!" after 1 second
```

The `.catch` Function

- If we're interested only in errors, then we can use `null` as the first argument: `.then(null, errorHandlingFunction)`.
- Or we can use `.catch(errorHandlingFunction)`, which is exactly the same:

```
1 let promise = new Promise((resolve, reject) => {
2   setTimeout(() => reject(new Error("Whoops!")), 1000);
3 });
4
5 // .catch(f) is the same as promise.then(null, f)
6 promise.catch(alert); // shows "Error: Whoops!" after 1 second
```

- The call `.catch(f)` is a complete analogue of `.then(null, f)`, it's just a shorthand.

On Settled Promises `then` Runs Immediately

- If a promise is pending, `.then/catch` handlers wait for the result. Otherwise, if a promise has already settled, they execute immediately:

```
1 // an immediately resolved promise
2 let promise = new Promise(resolve => resolve("done!"));
3
4 promise.then(alert); // done! (shows up right now)
```

- Some tasks may sometimes require time and sometimes finish immediately. The good thing is: the `.then` handler is guaranteed to run in both cases.

Handlers of `.then/.catch` Are Always Asynchronous

- Even when the Promise is immediately resolved, code which occurs on lines *below* your `.then/.catch` may still execute first.
- The JavaScript engine has an internal execution queue which gets all `.then/catch` handlers.
- But it only looks into that queue when the current execution is finished.
- In other words, `.then/catch` handlers are pending execution until the engine is done with the current code.

For Instance, Here:

```
1 // an "immediately" resolved Promise
2 const executor = resolve => resolve("done!");
3 const promise = new Promise(executor);
4
5 promise.then(alert); // this alert shows last (*)
6
7 alert("code finished"); // this alert shows first
```

- The promise becomes settled immediately, but the engine first finishes the current code, calls `alert`, and only *afterwards* looks into the queue to run `.then` handler.
- So the code *after* `.then` ends up always running *before* the Promise's subscribers, even in the case of an immediately-resolved Promise.
- Usually that's unimportant, but in some scenarios the order may matter a great deal.