Catching Synchronous Exceptions

In this lesson, we will see how to catch a synchronous exception.

WE'LL COVER THE FOLLOWING

Synchronous code an exception

Synchronous code is a traditional piece of code that is executed in the order that the lines are written. Asynchronous is when a piece of code is executed while the main thread (where the synchronous code is executed) occurs. In this lesson, we will focus on the exception that occurs in the main thread: synchronous exception.

Catching an exception

The `try` and `catch` structure handles exception like many well-known languages such as Java and C#. The `try` surrounds the code that is susceptible to throw an exception. The code executed in error will send the exception to the `catch` only if an exception occurs. The catch statement has a single parameter: the exception object. It's possible to provide a `finally` block after the `catch` that is executed every time, regardless of if there is an exception or not.

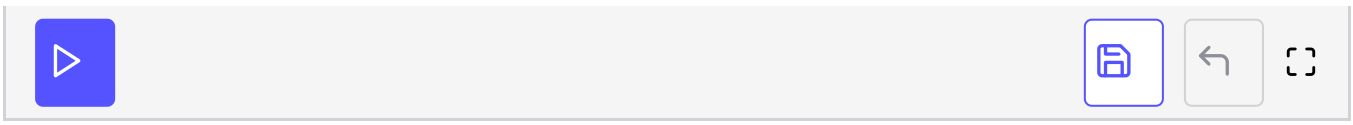
```
function throw1() {
  throw "error in string";
}

function throw2() {
  throw Error("Message Here");
```

```
}

function throw3() {
  const err: Error = { name: "Error", message: "Message" };
  throw err;
}

try {
  throw1();
} catch (e) {
  console.log("Exception 1", e); // String
}
try {
  throw2();
} catch (e) {
  console.log("Exception 2", e); // Full stack
}
try {
  throw3();
} catch (e) {
  console.log("Exception 2", e); // Object
}
```



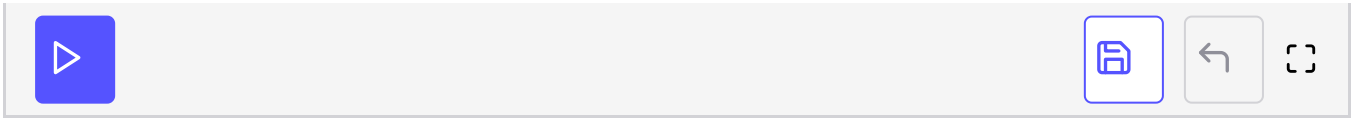
If you create a custom exception, it should inherit the type `Error` which gives the standard format with the minimum payload excepted. However, this is optional, and the exception mechanism will still work.

Narrowing exceptions with `instanceOf`

The object sent by the exception into the catch clause can use `instanceOf` to narrow a particular treatment depending on the type. However, in particular, it can be used to compare a type that inherits the base class's Error. This ensures that every constructor must set the prototype to the actual class to which it belongs. The reason is that since TypeScript 2.1, the constructor functions `Error`, `Array`, and `Map` don't propagate. The workaround is using the `setPrototypeOf` function in the constructor of each class in the hierarchy. The function sets the prototype to another object.

```
class ArgumentNullException extends Error {
    constructor(public name: string) {
        super("Argument was undefined");
        Object.setPrototypeOf(this, ArgumentNullException.prototype);
    }
}
class ReferenceException extends Error {
    constructor(public x: number, public y: number) {
        super("Reference was undefined");
```

```
        Object.setPrototypeOf(this, ReferenceException.prototype);
    }
}
function throwTwoExceptions(switcher: boolean) {
    if (switcher) {
        throw new ArgumentNullException("Switcher");
    }
    throw new ReferenceException(1, 2);
}
try {
    throw new ArgumentNullException("Switcher");
} catch (ex) {
    if (ex instanceof ArgumentNullException) {
        console.log("I can access name:" + ex.name);
    } else if (ex instanceof ReferenceException) {
        console.log("I can access x and y:" + ex.x + " and " + ex.y);
    }
}
```



At the moment, this is the only way to handle a custom error. It is not possible to catch errors directly with a type between parentheses.

```
try {
// ...
} catch(e: Error){ // Invalid!!!
// ...
}
```

Even the base `Error` type is not accepted in the `catch`.

Bubble up

Exceptions in JavaScript, like many other languages, when not handled will bubble up. This means that they will go up the stack until it is handled (caught) or until they reach the main thread and stop the application.

The following example calls three methods. **Line 13** is the invocation that will trigger the exception. By calling `method1`, the code goes into the method but then invokes `method2` which calls `method3`. The chain of invocation stops when `method3`, at **line 10** throws an exception. At that point, the code bubble up the exception. Because at **line 6** the code does not `catch`, the code continues to bubble up the exception.

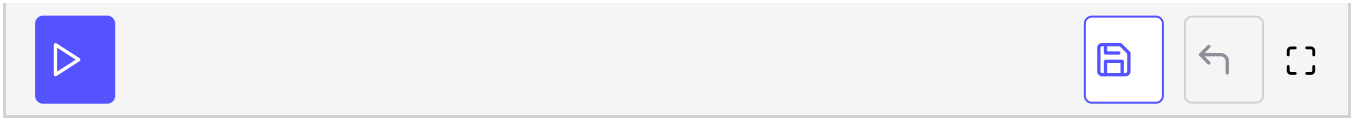
The bubbling up continues to bubble up the stack of invocation until a `catch`

The bubbling up continues to bubble up the stack of invocation until a `catch` is perceived at **line 14**.

```
function method1() { // Calls method2 which call method3 which throw an error
  method2();
}

function method2() { // Calls method3 which throw an error
  method3();
}

function method3() { // Calling this method will throw an error
  throw Error("Msg from method 3");
}
try {
  method1(); // Call method 1 that call method 2 that call method 3 that throw an error
} catch (e) {
  console.log(e.message);
}
```



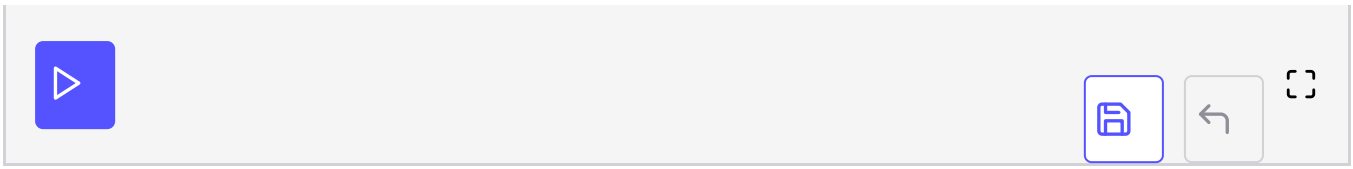
It is also possible to catch the exception and let it continue changing somewhere in the system to handle the exception completely.

The following example is an alteration of the previous one. This time, `method2` catch the exception from `method3` but `catch` it. Once it is caught, `method2` could have handled the exception and not continue to bubble up by doing nothing more. However, at **line 11**, the method is using `throw e` which resumes the bubbling up. Once again, **line 20** will `catch` the error.

```
function method1() {
  method2();
}

function method2() {
  try {
    method3();
  }
  catch (e) {
    console.log("Handled in method 2 but re-throw the ORIGINAL");
    throw e;
  }
}

function method3() {
  throw Error("Msg from method 3");
}
try {
  method1();
} catch (e) {
  console.log(e.message);
}
```



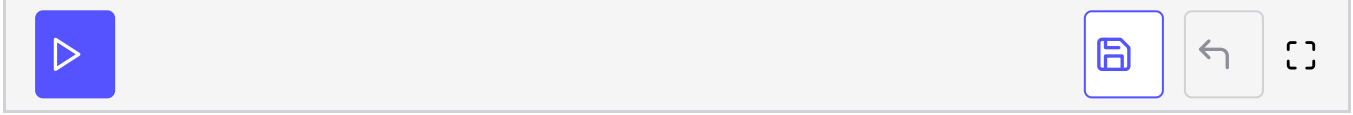
Bubbling up an exception can also be caught and return a new one. In the following example, the error returns a new error from `method2`. The original error could have been stored in a custom error as well to keep the original.

The following code, at **line 11** does not throw `e` as before but a new exception. **Line 20** `catch` and handle the new error, not the one from `method3`.

```
function method1() {
  method2();
}

function method2() {
  try {
    method3();
  }
  catch (e) {
    console.log("Handled in method 2 but re-throw the ORIGINAL");
    throw Error("New Error from Method 2");
  }
}

function method3() {
  throw Error("Msg from method 3");
}
try {
  method1();
} catch (e) {
  console.log(e.message);
}
```



Working with error with synchronous code is a matter of understanding when a `try-catch` block is needed. To find when it is required, a quick step-through the code give good insight. The next lesson will touch asynchronous code which is harder to grasp but as important.