JavaScript Handout

English edition

Contents

1 Getting started **3**
- 1.1 Your first program . . . 3
- 1.2 Comments & code style . . . 3

2 Values, variables & types **5**
- 2.1 let and const . . . 5
- 2.2 Numbers & maths . . . 5
- 2.3 Types & conversion . . . 6

3 Strings **7**
- 3.1 Characters & length . . . 7
- 3.2 String methods . . . 7
- 3.3 Template literals . . . 8

4 Decisions **10**
- 4.1 Booleans & comparisons . . . 10
- 4.2 if / else if / else . . . 11
- 4.3 switch . . . 12

5 Loops **13**
- 5.1 for loops . . . 13
- 5.2 while loops . . . 14
- 5.3 for...of, break & continue . . . 14

6 Functions **16**
- 6.1 Parameters & return . . . 16
- 6.2 Arrow functions . . . 16
- 6.3 Scope . . . 17

7 Arrays **18**
- 7.1 Making & changing arrays . . . 18
- 7.2 map, filter & reduce . . . 18
- 7.3 Searching & sorting . . . 19

8 Objects **21**
- 8.1 Objects & properties . . . 21
- 8.2 Arrays of objects . . . 21
- 8.3 Looping over an object . . . 22

9 Errors **23**
- 9.1 Reading error messages . . . 23
- 9.2 try, catch & throw . . . 24

1 Getting started

1.1 ■ Your first program

A program is a list of instructions for the computer. In JavaScript, each instruction is a statement, and the computer runs the statements from top to bottom.

`console.log(...)` prints a value in the console, the text area under the code:

```
console.log("Hello, world!");
console.log(7 * 6);   // 42
```

- Text goes inside quotes: `"Hello"` or `'Hello'`. Numbers do not.
- End each statement with a semicolon `;`. JavaScript can often guess where one is missing, but writing it is clearer.
- JavaScript is case-sensitive: `console.log` works, `Console.Log` does not.

`console.log` can print several values at once. It puts one space between them:

```
console.log("Total:", 3 + 4);   // Total: 7
console.log(1, 2, 3);           // 1 2 3
```

1.2 ■ Comments & code style

A comment is a note for people. JavaScript ignores it when the program runs.

```
// A one-line comment starts with two slashes.
console.log("This line runs");   // a comment can follow code

/* A block comment
   can cover several lines. */
console.log("So does this one");
```

Good style makes code easy to read:

- Indent the lines inside `{ }` by two or four spaces, and do it the same way every time.
- Give variables names that say what they hold: `total`, not `t`.
- JavaScript names use camelCase: `firstName`, `scoreTotal`.

```
// Work out the area of a rectangle
const width = 4;
const height = 3;
console.log(width * height);   // 12
```

Common mistakes

- Text needs matching quotes: `"Hi'` is an error.
- `console.log` is all lower case: `Console.log` and `console.Log` are errors.
- Everything after `//` on a line is ignored, so `// console.log(x)` prints nothing.

2 Values, variables & types

2.1 ■ let and const

A variable is a named box that holds a value. Make one with `let` or `const`, then use its name:

```
let score = 10;
score = score + 5;      // let: the value can change
console.log(score);     // 15

const name = "Mei";     // const: the name cannot be reassigned
console.log(name);      // Mei
```

- `=` means **assign**: work out the right side, then store it in the box on the left.
- Use `const` by default. Use `let` only when the value must change later.
- Reassigning a `const` is an error: `name = "Tom"` is not allowed. A const object or array can still have its contents changed.

2.2 ■ Numbers & maths

These examples use the number type for whole and decimal values. BigInt is a separate integer type. The arithmetic operators are:

Operator	Meaning	Example	Result
`+ -`	add, subtract	`7 - 2`	5
`* /`	multiply, divide	`7 / 2`	3.5
`%`	remainder	`7 % 2`	1
`**`	power	`2 ** 3`	8

```
console.log(7 / 2);              // 3.5
console.log(Math.floor(7 / 2));  // 3   (whole-number division)
console.log(7 % 2);              // 1   (odd numbers leave 1)
console.log(2 + 3 * 4);          // 14  (* before +)
console.log((2 + 3) * 4);        // 20
```

`Math` holds useful functions:

```
console.log(Math.round(2.5));    // 3
console.log(Math.max(4, 9, 2)); // 9
console.log(Math.sqrt(49));      // 7
```

- `Math.random()` gives a random number from 0 up to (not including) 1. `Math.floor(Math.random() * 6) + 1` is a dice roll from 1 to 6.
- Decimals are stored in binary, so `0.1 + 0.2` prints `0.30000000000000004`. Round a result before you show it: `(0.1 + 0.2).toFixed(2)` is `"0.30"`.

2.3 ■ Types & conversion

Every value has a type. `typeof` tells you which:

```
console.log(typeof 42);         // number
console.log(typeof "42");       // string
console.log(typeof true);       // boolean
console.log(typeof undefined);  // undefined
```

- A variable made with `let x;` and no value is `undefined`.
- `null` means "no value", set on purpose.

`+` with a string **joins** instead of adding, so convert text to a number first:

```
console.log("5" + 1);            // 51  (joined)
console.log(Number("5") + 1);    // 6   (added)
console.log(parseInt("12px"));   // 12  (reads the number at the start)
console.log(String(9) + "!");    // 9!
```

- A text that is not a number gives `NaN` ("not a number"): `Number("abc")` is `NaN`.
- `-`, `*` and `/` convert text to a number by themselves (`"6" * 2` is `12`), but do not rely on it: convert on purpose.

Common mistakes

- `=` assigns; it does not compare. Comparing uses `===` (topic 4).
- `"5" + 1` is `"51"`. Convert input with `Number(...)` before doing maths with it.
- `Math.round(2.5)` is `3`, but `Math.round(-2.5)` is `-2`: it rounds halves up, towards the larger number.

3 Strings

3.1 ■ Characters & length

A string is text inside quotes. With simple letters, each character has one position, its index ; the first index is 0. JavaScript counts UTF-16 code units: an emoji may take two positions.

```
const word = "Script";
console.log(word.length);      // 6
console.log(word[0]);          // S
console.log(word[2]);          // r
console.log(word.at(-1));      // t  (counts back from the end)
console.log(word[word.length - 1]);   // t
```

- The last index is `length - 1`. `word[6]` is `undefined`, not an error.
- Strings are immutable : `word[0] = "s"` cannot change the string and throws a TypeError in strict mode. Build a new string instead.

Join strings with `+`, which is called concatenation :

```
const first = "Ada";
const last = "Lovelace";
console.log(first + " " + last);   // Ada Lovelace
```

3.2 ■ String methods

A method is a function that belongs to a value. Call it with a dot. A string method gives back a **new** value and leaves the original alone.

```
const city = "  Hong Kong ";
console.log(city.trim());                  // Hong Kong
console.log(city.trim().toUpperCase());    // HONG KONG
console.log(city.length);                  // 12  (the spaces are still
↪  there)
```

Method	Meaning	Example	Result
`.toUpperCase()` / `.toLowerCase()`	change case	`"Hi".toLowerCase()`	`"hi"`
`.trim()`	remove spaces at both ends	`" hi ".trim()`	`"hi"`
`.includes(t)`	does it contain `t`?	`"cat".includes("a")`	`true`
`.indexOf(t)`	first position of `t`, or `-1`	`"cat".indexOf("t")`	`2`
`.slice(a, b)`	characters from `a` up to (not including) `b`	`"Script".slice(0, 3)`	`"Scr"`
`.split(sep)`	break into an array	`"a,b,c".split(",")`	`["a","b","c"]`
`.replace(a, b)`	swap the first `a` for `b`	`"cat".replace("c", "b")`	`"bat"`

```
const email = "mei@school.org";
const at = email.indexOf("@");
console.log(email.slice(0, at));          // mei
console.log(email.slice(at + 1));         // school.org
console.log(email.split("@"));            // ["mei","school.org"]
```

3.3 ■ Template literals

A template literal is a string written between backticks (on most keyboards, the key left of 1). Put any expression inside `${ }` and its value is placed into the text:

```
const name = "Mei";
const age = 17;
console.log(`${name} is ${age} years old.`);   // Mei is 17 years old.
console.log(`Next year: ${age + 1}`);          // Next year: 18
```

- Backticks can also hold several lines of text.
- This is easier to read than many `+` signs: compare `"Hi " + name + "!"` with `` `Hi ${name}!` ``.

```
const price = 4.5;
console.log(`Two cost $${(price * 2).toFixed(2)}`);   // Two cost $9.00
```

Common mistakes

- `${ }` only works inside backticks. In `"Hi ${name}"` the text is printed exactly as typed.
- Methods do not change the string: `city.trim()` on its own line does nothing. Store the result: `const clean = city.trim();`.
- `slice(a, b)` stops **before** `b`, just like the last index is `length - 1`.

4 Decisions

4.1 ■ Booleans & comparisons

A Boolean is either `true` or `false`. A comparison gives a Boolean:

Operator	Meaning	Example	Result
`===`	equal (same value and same type)	`5 === 5`	`true`
`!==`	not equal	`5 !== 3`	`true`
`< >`	less than, greater than	`2 > 7`	`false`
`<= >=`	at most, at least	`7 >= 7`	`true`

```
console.log(5 === 5);     // true
console.log(5 === "5");   // false  (a number is not a string)
console.log(5 == "5");    // true   (== converts first: avoid it)
```

- Always use `===` and `!==`. The older `==` converts types first and gives surprising answers.

For Boolean conditions, use the logical operators below. With other values, `&&` and `||` return one of their operands; they do not always return a Boolean:

- `&&` (and): true only if **both** sides are true.
- `||` (or): true if **at least one** side is true.
- `!` (not): turns `true` into `false` and back.

```
const age = 15;
console.log(age >= 13 && age <= 19);   // true   (a teenager)
console.log(age < 5 || age > 65);      // false
console.log(!(age > 18));              // true
```

4.2 ■ if / else if / else

An `if` statement runs a block of code when its condition is truthy (treated as true). With Boolean conditions, this means the value `true`. `else if` tests another condition; `else` catches everything left:

```
const mark = 72;
if (mark >= 80) {
  console.log("A");
} else if (mark >= 60) {
  console.log("B");
} else {
  console.log("C");
}
// prints B
```

- JavaScript checks the conditions from the top and runs **only the first** true branch .
- The condition goes in round brackets; the code to run goes in curly braces `{ }`.

So the order matters. Put the strictest test first:

```
const temp = 35;
if (temp > 30) {
  console.log("hot");
} else if (temp > 20) {
  console.log("warm");   // skipped: the first branch already ran
}
// prints hot
```

4.3 ■ switch

A `switch` compares one value against several fixed cases. It is neater than a long `else if` chain when every test is "is it equal to …?":

```
const day = "Sat";
switch (day) {
  case "Sat":
  case "Sun":
    console.log("weekend");
    break;
  case "Fri":
    console.log("nearly there");
    break;
  default:
    console.log("school day");
}
// prints weekend
```

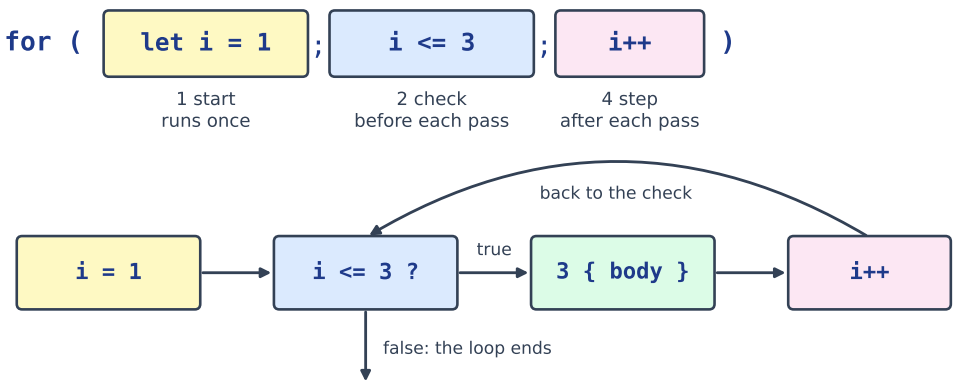
- `switch` compares with `===`, so `case 1:` does not match the string `"1"`.
- `break` ends the `switch`. Without it, the code **falls through** into the next case.
- `default` runs when no case matches, like a final `else`.
- Two cases with no code between them share the same code, as `"Sat"` and `"Sun"` do here.

Common mistakes

- `if (x = 5)` assigns instead of comparing, and the test is then always true. Write `if (x === 5)`.
- `if (age > 13 && < 19)` is an error: repeat the variable, `age > 13 && age < 19`.
- A missing `break` in a `switch` runs the next case too.

5 Loops

5.1 ■ for loops



A loop repeats a block of code. A `for` loop is best when you know how many times. Its brackets hold three parts, separated by semicolons:

```
for (let i = 1; i <= 3; i++) {
  console.log(i);
}
// 1
// 2
// 3
```

1. **start**: `let i = 1` runs once, before the loop.
2. **condition**: `i <= 3` is checked before every pass; the loop stops when it is false.
3. **step**: `i++` (add 1 to `i`) runs after every pass.

Each pass through the loop is one iteration . Change the step to count in other ways:

```
let line = "";
for (let i = 10; i > 0; i -= 3) {
  line += i + " ";
}
console.log(line);   // 10 7 4 1
```

The accumulator pattern makes a variable **before** the loop and updates it in every pass:

```
let total = 0;
for (let n = 1; n <= 5; n++) {
  total += n;          // the same as total = total + n
}
console.log(total);    // 15
```

5.2 ■ while loops

A `while` loop repeats while its condition is true. Use it when you do not know in advance how many passes you need:

```
let money = 100;
let years = 0;
while (money < 200) {
  money = money * 1.1;   // 10% interest each year
  years++;
}
console.log(years);      // 8
```

- The condition is checked **before** each pass, so the body may run zero times.
- If the condition stays true and no `break` or other exit is reached, you get an infinite loop . (Press **Stop** if that happens.)

5.3 ■ for...of, break & continue

`for...of` visits every item of an array (or every character of a string) in turn, with no index to manage:

```
const fruits = ["apple", "kiwi", "plum"];
for (const fruit of fruits) {
  console.log(fruit.toUpperCase());
}
// APPLE
// KIWI
// PLUM
```

`break` leaves a loop at once; `continue` skips the rest of this pass and goes on to the next one:

```
const marks = [55, 80, -1, 90];
let sum = 0;
for (const m of marks) {
  if (m < 0) {
    break;            // -1 means "end of the data"
  }
  if (m < 60) {
    continue;         // skip a fail, keep looping
  }
  sum += m;
}
console.log(sum);     // 80
```

Common mistakes

- An off-by-one error: `i < 3` runs for 0, 1, 2; `i <= 3` runs one more time. Count the passes.
- Starting the total inside the loop: `let total = 0;` must come **before** the loop, or it is reset every pass.
- A `while` loop runs for ever if its condition stays true and no exit is reached.

6 Functions

6.1 ■ Parameters & return

A function is a reusable piece of code. A function declaration gives it a name, which you can use to call it:

```
function greet(name) {
  return `Hello, ${name}!`;
}

console.log(greet("Mei"));   // Hello, Mei!
console.log(greet("Tom"));   // Hello, Tom!
```

- `name` is a parameter : a variable that receives a value when the function is called.
- `"Mei"` is the argument : the actual value passed in.
- `return` sends a value back to the caller and ends the function at once.

A regular function that reaches its end without returning a value gives back `undefined`. So **return** the answer, and let the caller decide whether to print it:

```
function area(width, height) {
  return width * height;
}

const a = area(4, 3);
console.log(a + 1);          // 13  (a returned value can be used again)
```

6.2 ■ Arrow functions

An arrow function is a shorter way to write a function. Store it in a `const`:

```
const square = (n) => n * n;
const add = (a, b) => a + b;

console.log(square(5));      // 25
console.log(add(2, 3));      // 5
```

- With one expression after `=>`, its value is returned without writing `return`.
- With curly braces you need `return` again:

```
const grade = (mark) => {
  if (mark >= 50) {
    return "pass";
  }
  return "fail";
};
console.log(grade(64));      // pass
```

Arrow functions are often passed to other functions, as you will see with `map` and `filter` in topic 7.

6.3 ■ Scope

The scope of a variable is the part of the code that can see it. A variable made with `let` or `const` inside `{ }` is accessible in that block and its nested scopes. A nested function can keep access after the outer call returns:

```
const level = "global";

function show() {
  const inside = "local";
  console.log(level);        // global  (outer variables are visible)
  console.log(inside);       // local
}

show();
console.log(typeof inside);  // undefined  (inside does not exist out
↪  here)
```

- A variable made inside a function is a local variable ; each call gets a fresh one.
- Pass values **in** as parameters and get results **out** with `return`, instead of changing outer variables from inside a function.

Common mistakes

- Forgetting `return`: the function works out the answer, then gives back `undefined`.
- Calling without brackets: `greet` is the function itself; `greet("Mei")` runs it.
- Printing inside the function **and** printing its result prints twice, or prints `undefined`.

7 Arrays

7.1 ■ Making & changing arrays

An array holds a list of values in order, inside square brackets. Each item is an element , and the first index is `0`:

```
const scores = [72, 88, 95];
console.log(scores[0]);        // 72
console.log(scores.length);    // 3
scores[1] = 90;                // change one element
console.log(scores);           // [72,90,95]
```

- `const` stops you giving `scores` a whole new array, but you can still change what is inside it.
- Reading past the end gives `undefined`: `scores[3]` is `undefined`.

Add and remove at the end with `push` and `pop`:

```
const queue = ["Ann", "Ben"];
queue.push("Cai");             // add to the end
console.log(queue);            // ["Ann","Ben","Cai"]
const last = queue.pop();      // remove from the end and return it
console.log(last);             // Cai
console.log(queue.length);     // 2
```

An array can hold arrays. That makes a grid, read with two indexes, row first:

```
const grid = [
  [1, 2, 3],
  [4, 5, 6],
];
console.log(grid[1][2]);       // 6  (row 1, column 2)
```

7.2 ■ map, filter & reduce

These methods take a function and apply it to the array elements. They return a **new** array (or value) without themselves replacing the original items. The function you pass in can still change the original array or its objects:

- `map` makes a new array by changing every element.
- `filter` keeps only the elements for which the function returns `true`.
- `reduce` combines all the elements into one value, such as a total.

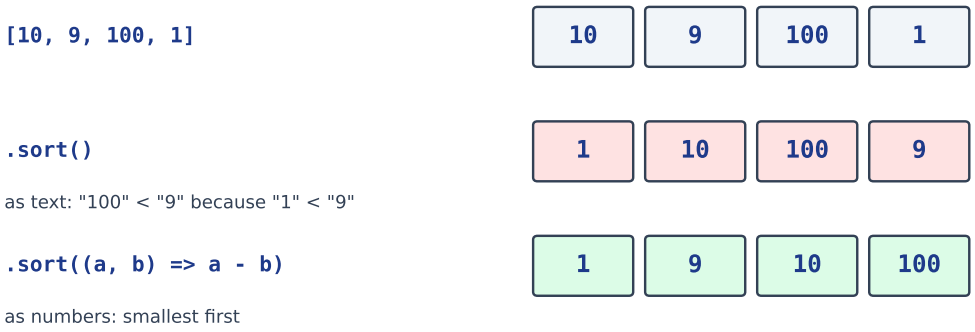
```
const prices = [4, 10, 25];
const withTax = prices.map((p) => p * 1.1);
console.log(withTax.map((p) => p.toFixed(2)));   //
↪  ["4.40","11.00","27.50"]

const cheap = prices.filter((p) => p < 20);
console.log(cheap);                              // [4,10]

const total = prices.reduce((sum, p) => sum + p, 0);
console.log(total);                              // 39
```

In `reduce`, `sum` is the accumulator. It starts at the last argument (`0`) and takes the function's result after each element.

7.3 ■ Searching & sorting



Method	Gives back	Example with `[5, 8, 2]`	Result
`.includes(x)`	`true` or `false`	`.includes(8)`	`true`
`.indexOf(x)`	the first index of `x`, or `-1`	`.indexOf(2)`	`2`
`.find(f)`	the first element where `f` is true	`.find((n) => n > 6)`	`8`
`.findIndex(f)`	its index, or `-1`	`.findIndex((n) => n > 6)`	`1`

`sort` puts the elements in order. With no function, it sorts them **as text**, so numbers come out in the wrong order:

```
const nums = [10, 9, 100, 1];
console.log([...nums].sort());                   // [1,10,100,9]   (as
↪  text!)
console.log([...nums].sort((a, b) => a - b));    // [1,9,10,100]
console.log([...nums].sort((a, b) => b - a));    // [100,10,9,1]
```

- The compare function returns a negative number when `a` should come first, and a positive number when `b` should.
- `sort` changes the array itself. `[...nums]` makes a copy first, so `nums` keeps its order.

Common mistakes

- The last element is `arr[arr.length - 1]`, not `arr[arr.length]`.
- `map` and `filter` do not change the array: store their result.
- `nums.sort()` on numbers sorts them as text. Pass `(a, b) => a - b`.

8 Objects

8.1 ■ Objects & properties

An object groups related values under names. Each name-and-value pair is a property. The name is called the key:

```
const pupil = {
  name: "Mei",
  age: 17,
  form: "11A",
};
console.log(pupil.name);       // Mei
console.log(pupil["form"]);    // 11A  (bracket form, with the key as a
↪  string)
```

- Read a property with a dot (`pupil.age`) or with brackets (`pupil["age"]`). Use brackets when the key is in a variable: `pupil[field]`.
- A key that does not exist gives `undefined`.

Change a property, or add a new one, by assigning to it:

```
const pupil = { name: "Mei", age: 17 };
pupil.age = 18;                // change
pupil.house = "Red";           // add
console.log(pupil);            // {"name":"Mei","age":18,"house":"Red"}
```

8.2 ■ Arrays of objects

A list of records is an array of objects. Combine it with the array methods from topic 7:

```
const pupils = [
  { name: "Mei", mark: 88 },
  { name: "Tom", mark: 64 },
  { name: "Sara", mark: 95 },
];

const passed = pupils.filter((p) => p.mark >= 70);
console.log(passed.map((p) => p.name));          // ["Mei","Sara"]

const best = [...pupils].sort((a, b) => b.mark - a.mark)[0];
console.log(`${best.name} has the top mark`);      // Sara has the top
↪  mark
```

To total a property, use `reduce` or a loop:

```
const pupils = [{ mark: 88 }, { mark: 64 }, { mark: 95 }];
let sum = 0;
for (const p of pupils) {
  sum += p.mark;
}
console.log((sum / pupils.length).toFixed(1));      // 82.3
```

8.3 ■ Looping over an object

`Object.keys` gives an array of the keys, and `Object.entries` gives `[key, value]` pairs, so you can loop over them:

```
const stock = { pens: 12, rulers: 0, books: 5 };

console.log(Object.keys(stock));             // ["pens","rulers","books"]

for (const [item, count] of Object.entries(stock)) {
  if (count === 0) {
    console.log(`${item}: sold out`);
  }
}
// rulers: sold out
```

An object is a good counter: use each thing you count as a key.

```
const words = ["red", "blue", "red", "red"];
const counts = {};
for (const w of words) {
  counts[w] = (counts[w] || 0) + 1;   // 0 the first time the key is seen
}
console.log(counts);                  // {"red":3,"blue":1}
```

Common mistakes

- `pupil.field` looks for a key called `field`. When the key is in a variable, write `pupil[field]`.
- A comma goes between properties, not a semicolon: `{ name: "Mei", age: 17 }`.
- `for...of` cannot loop over an object directly. Loop over `Object.keys(...)` or `Object.entries(...)`.

9 Errors

9.1 ■ Reading error messages

When JavaScript cannot carry on, it stops and shows an error . The first word names the kind of error; the rest says what went wrong and where. Read it, then look at the line it names.

```
SyntaxError: missing ) after argument list
ReferenceError: totl is not defined
TypeError: Cannot read properties of undefined (reading 'length')
```

Error	What it means	A typical cause
`SyntaxError`	the code is not written correctly, so nothing runs	a missing bracket or quote
`ReferenceError`	a name that does not exist	a spelling mistake : `totl` for `total`
`TypeError`	the value is the wrong type for what you did	using `.length` on `undefined`

The exact wording can differ between browsers, but the name of the error is the same. This program catches each error so that it can print its name:

```
try {
  const total = 5;
  console.log(totl);
} catch (e) {
  console.log(e.name);            // ReferenceError
}

try {
  const pupil = undefined;
  console.log(pupil.length);
} catch (e) {
  console.log(e.name);            // TypeError
}
```

Some mistakes give no error at all, just a wrong answer: a logic error . To find one, print the values along the way with `console.log`, and check them by hand.

9.2 ■ try, catch & throw

Code that might fail goes in a `try` block. If an error happens there, JavaScript jumps to `catch` instead of stopping, and `e` holds the error:

```
try {
  JSON.parse("{not valid}");
  console.log("parsed");          // skipped
} catch (e) {
  console.log("Could not read the data");
}
console.log("The program carries on");
```

Your own code can raise an error with `throw`. This lets a function refuse bad input clearly, instead of returning a wrong answer:

```
function average(list) {
  if (list.length === 0) {
    throw new Error("the list is empty");
  }
  return list.reduce((s, x) => s + x, 0) / list.length;
}

console.log(average([4, 6]));     // 5
try {
  average([]);
} catch (e) {
  console.log(`Error: ${e.message}`);   // Error: the list is empty
}
```

- `new Error("…")` makes an error whose `message` is your text.
- Only use `try` around code that can really fail, such as reading data. Do not use it to hide mistakes in your own code.

Common mistakes

- Skipping the message. Its **first** line names the kind of error and what went wrong; start there.
- A `catch` that does nothing (`catch (e) {}`) hides the problem: at least print `e.message`.
- `throw "text"` works, but `throw new Error("text")` also records where the error happened.