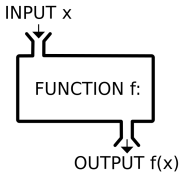
TypeScript Functions (& Lambdas)



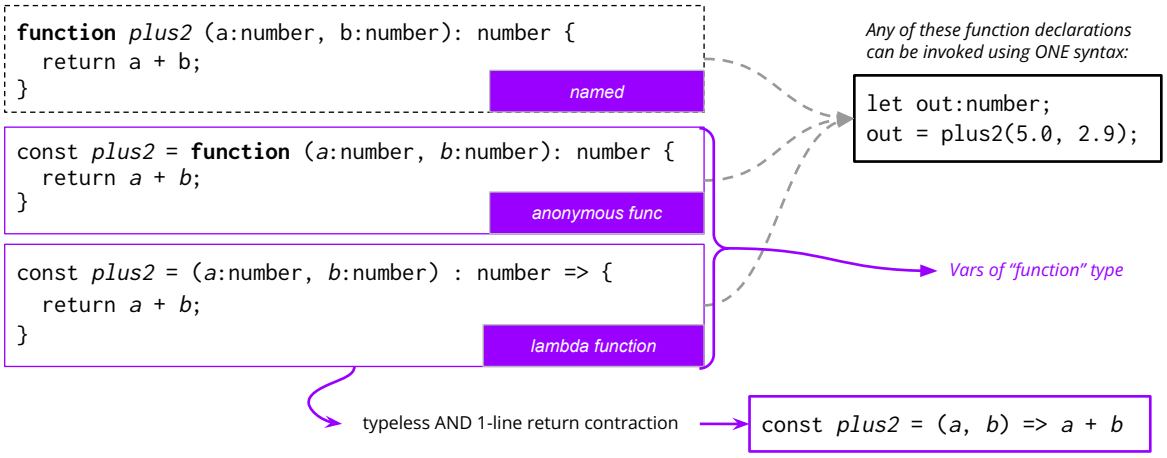
Topics Covered

- Methods vs. standalone functions
- ***Function as a type***
- Anonymous and “fat” arrow functions (lambdas)
- High-Order Functions
  - Functions that take another function as input parameter
- Array manipulation functions

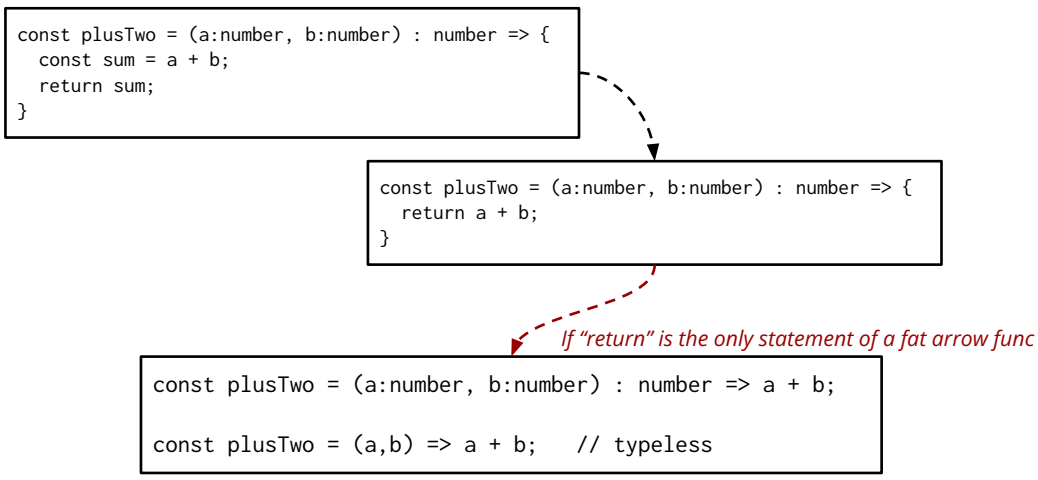
Important Takeaway Concept #1:
JS & TS allow variables of type Function

Important Takeaway Concept #2:
JS & TS variables can hold either *data* or *code*

Three variations of Function Declarations



Fat Arrow fns: single-line `return` contraction



VSCode Demo / Replit Playground

Fat arrow funcs do not hide `this`
(when used in a class)

Variables of func type

```
const plus20 = “+20”;
const plus22 = { positive: true, value: 22 }
```

plus20 and plus22 are variables that hold your DATA

```
const plus2 = function (a:number, b:number): number {
  return a + b;
}

const plusTwo = (a:number, b:number) : number => {
  return a + b;
}
```

```
console.log(typeof plus20); // string
console.log(typeof plus22); // object

console.log(typeof plus2);   // function
console.log(typeof plusTwo); // function
```

plus2 and plusTwo are variables that hold your CODE

Functions as Arguments (to another Fn)

Array.sort()

```
const atoms = [“Neon”, “Iron”, “Calcium”, “Hydrogen”]
console.log(atoms.sorted())         // [“Calcium”, “Hydrogen”, “Iron”, “Neon”

const primes = [23, 17, 5, 101, 19]
const sorted_nums = primes.sort()
console.log(sorted_nums)            // [101, 17, 19, 23, 5] What???
```

Replit Playground

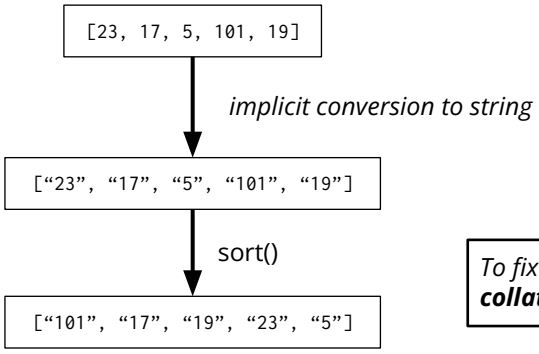
Array.prototype.sort()

MDN online doc

The `sort()` method sorts the elements of an array *in place* and returns the reference to the same array, now sorted. The default sort order is ascending, built upon converting the elements into strings, then comparing their sequences of UTF-16 code units values.

Online Doc

Array sort() builtin behavior



To fix this “bug”, we have to tell sort() the ***collating order between two data items***

Array.sort() with collating order

```
function numericOrder(a:number, b:number): number {
  if (a < b) return -7121;          // any negative number
  else if (a > b) return +8321;     // any positive number
  else return 0;
}

const primes = [23, 17, 5, 101, 19]
const sorted_nums = primes.sort(numericOrder)
console.log(sorted_nums)       // [5, 17, 19, 23, 101]   Ok
```

Replit Playground

*The collating function **must return a number***

- *Negative when the "first" item should be placed BEFORE the "second" item*
- *Positive when the "first" item should be placed AFTER the "second" item*
- *Zero when the order of the two items is irrelevant*

Array.sort() on objects

```
type Language = {
  name: string;  yearCreated: number
}

const langs: Language[] = [
  { name: "C",          yearCreated: 1970},
  { name: "JavaScript", yearCreated: 1995},
  { name: "Fortran",    yearCreated: 1954}]

function orderByName(a:Language, b:Language): number {
  return a.name.localeCompare(b.name)
}

function orderByYear(a:Language, b:Language): number {
  return a.yearCreated - b.yearCreated
}

langs.sort(orderByName)
```

Replit Playground

The collating function takes two parameters of type Language *but* ***must return a number***

Array.sort() on objects: incorrect collating funcs

```
type Language = {
  name: string;  yearCreated: number
}

const langs: Language[] = [
  { name: "C",          yearCreated: 1970},
  { name: "JavaScript", yearCreated: 1995},
  { name: "Fortran",    yearCreated: 1954}]

function orderByName(a:string, b:string): number {
  return a.localeCompare(b)
}

function orderByYear(a:number, b:number): number {
  return a - b
}

langs.sort(orderByName)
```

The collating function must take two parameters of type Language

Collating Functions: named, unnamed, lambda

```
type Language = {
  name: string;  yearCreated: number
}

const langs: Language[] = [
  { name: "C",          yearCreated: 1970 },
  { name: "JavaScript", yearCreated: 1995 },
  { name: "Fortran",    yearCreated: 1954 }
]
```

Option 1: named function

```
function orderByName(a:Language, b:Language): number {
  return a.name.localeCompare(b.name)
}
langs.sort(orderByName)
```

Replit Playground

Option 2: unnamed function

```
langs.sort(
  function (a:Language, b:Language): number {
    return a.name.localeCompare(b.name)
  }
)
```

Option 3: lambda function

```
langs.sort(
  (a:Language, b:Language): number => {
    return a.name.localeCompare(b.name)
  }
)
```

Opt 4: typeless lambda & 1-line return contraction

```
langs.sort(
  (a, b) => a.name.localeCompare(b.name)
)
```

Function Optional Parameters/Arguments

```
// whoAmI can be called with 2, 3, or 4 args
const whoAmI = (name: string, age: number,  occupation?: string, spouse?: string): void => {
  console.log("Work as", occupation);
  console.log("Spouse name:", spouse ?? "N/A")
}
```

```
whoAmI("Andy", 22);                          // Work as undefined
                                             // Spouse name: N/A
whoAmI("Bob", 43, "banker");                 // Work as banker
                                             // Spouse name: N/A
whoAmI("Chuck", 31, undefined, "Cindy"); // Work as undefined
                                             // Spouse name Cindy
whoAmI("Chuck", 31, null, "Cindy");          // Work as null
                                             // Spouse name Cindy
```


Function Parameter Default Value

```
const whoAmI = (name: string, age: number,
                  occupation: string = "Student",
                  spouse?: string): void => {
  console.log("Work as", occupation);
  console.log("Spouse name:", spouse ?? "N/A")
}
```

Undefined implies "skip" this arg

```
whoAmI("Andy", 22);                          // Work as Student
                                             // Spouse name: N/A
whoAmI("Bob", 43, "banker");                 // Work as banker
                                             // Spouse name: N/A
whoAmI("Chuck", 31, undefined, "Cindy"); // Work as Student
                                             // Spouse name Cindy
whoAmI("Chuck", 31, null, "Cindy");          // Work as null
                                             // Spouse name Cindy
```

Array Operations

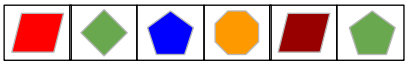
Array high-order functions

- Array.every(), Array.some()
- Array.find(), findIndex()
- Array.filter(), Array.map(), Array.flatmap()
- Array.forEach()
- Array.reduce()
- … and many others
- flatMap() is available in ES2019

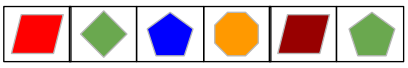
```
// tsconfig.json
{
  “compilerOptions”: {
    “target”: “ES2019”,
    // other options go here
  }
```

```
type Shape = {
  color: string;
  numSides: number
  sideDims: Array<number> // the length of each side
}
```

```
let shapes: Array<Shape> = [_____]
```



Array.some(): do we have any green shape?



shapes.some (?????) → YES

```
function isGreen(s: Shape): boolean {
  return s.color === “green”
}

let shapes:Array<Shape> = [_____];
const someGreen = shapes.some(isGreen);
console.log(someGreen);  // true
```

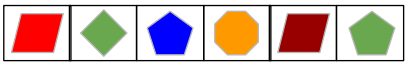
```
const someGreen = shapes.some(function(s: Shape): boolean {
                                return s.color === “green”
                              });
```
Anonymous func

```
const someGreen = shapes.some((s: Shape): boolean => {
                                return s.color === “green”
                              });
```
Anon. fat arrow

```
const someGreen = shapes.some((s: Shape): boolean =>
                                s.color === “green”);
```
1-line return elimination

```
const someGreen = shapes.some((s) =>  s.color === “green”);
```
No explicit type

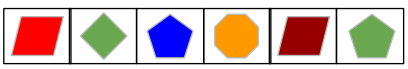
Array.some(): do we have any green? (Incorrect!!!)



shapes.some (?????) → YES

```
// isGreen must take a Shape as its input parameter
// NOT a string!!!
function isGreen(col: string): boolean {
  return col === “green”
}

let shapes:Array<Shape> = [_____];
const someGreen = shapes.some(isGreen);
console.log(someGreen);  // true
```


Array.forEach(): inspect all shapes



shapes.forEach (????)

```
function printShape(s: Shape): void{
  console.log(“# of sides”, s.sides);
}

let shapes:Array<Shape> = [_____];
shapes.forEach(printShape);
```

```
shapes.forEach(function(s: Shape): void {
                 console.log(“# of sides”, s.sides);
               });
```

Anonymous func

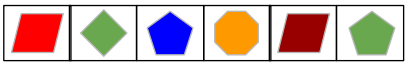
```
shapes.forEach((s: Shape): void => {
                 console.log(“# of sides”, s.sides);
               });
```

Anon. fat arrow

```
shapes.forEach((s) =>  {
                  console.log(“@ of sides:, s.sides);
               });
```

No explicit type

Array.every(): are all shapes pentagon?



shapes.every (?????) → NO

```
function isPenta(s: Shape): boolean {
  return s.sides === 5
}

let shapes:Array<Shape> = [_____];
const allPenta= shapes.every(isPenta);
console.log(allPenta);  // false
```

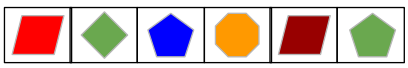
```
const allPenta = shapes.every(function(s: Shape): boolean {
                                return s.sides === 5
                              });
```
Anonymous func

```
const allPenta = shapes.every((s: Shape): boolean => {
                                return s.sides === 5
                              });
```
Anon. fat arrow

```
const allPenta = shapes.every((s: Shape): boolean =>
                                s.sides === 5)
```
1-line return simplification

```
const allPenta = shapes.every((s) =>  s.sides === 5);
```
No explicit type

Array.findIndex(): where is XXX?



shapes.findIndex(isPenta) → At pos 2

shapes.findIndex(isTriang) → -1

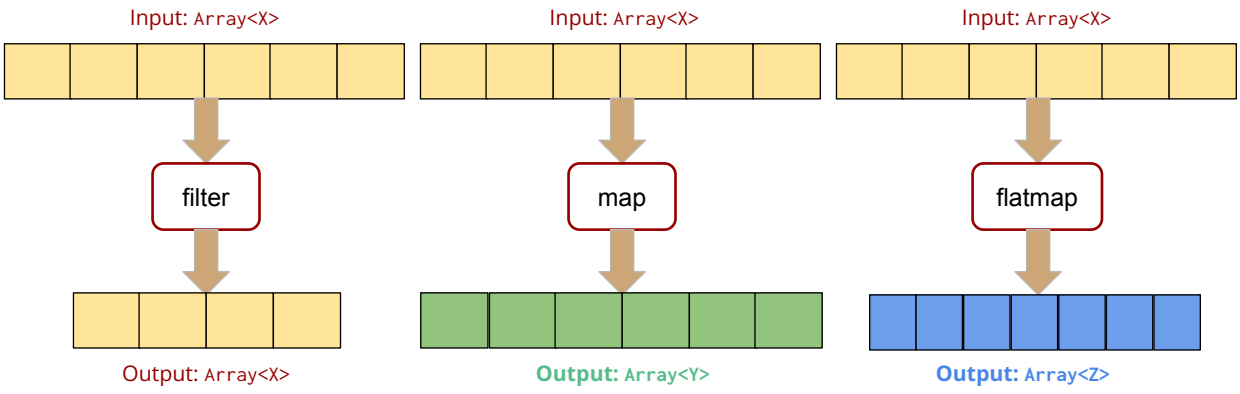
```
function isPenta(s: Shape): boolean {
  return s.sides === 5
}

let shapes:Array<Shape> = [_____];
const pent = shapes.findIndex(isPenta);
```

```
const pent= shapes.findIndex((s: Shape): boolean => {
                                return s.sides === 5
                              });
```
Anon. fat arrow

```
const pent= shapes.findIndex((s: Shape): boolean =>
                                s.sides === 5)
```
1-line return simplification

```
const pent= shapes.findIndex((s) =>  s.sides === 5);
const triPos = shapes.findIndex(s => s.sides === 3)
```
No explicit type

Coding Demo

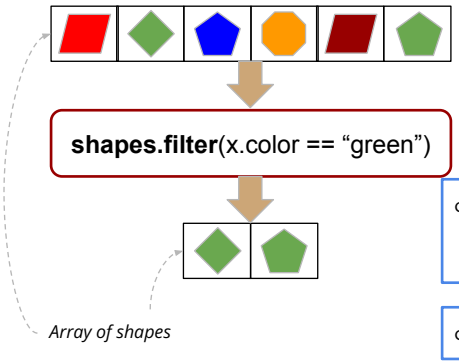


*new array of the **original type** (with **fewer** elements copies of original)*

*new array of a **new type**, same length as the original*

*new array of a **new type** **more** elements than original*

Array.filter(): give me only green shapes

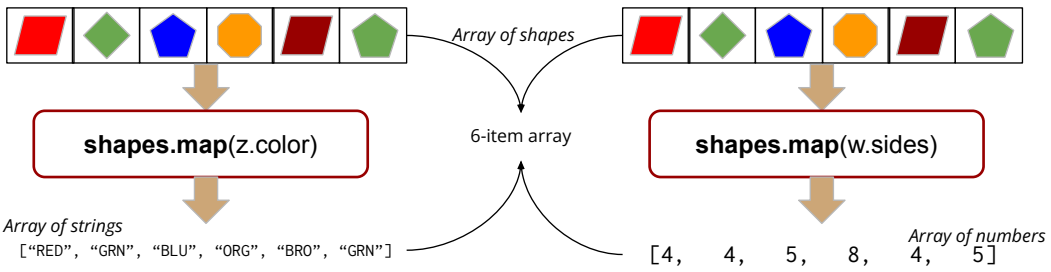



```
const shapes: Shape[];

function isGreen (x:Shape): boolean {
  return x.color === “green”
}
const greenOnly: Shape[] = shapes.filter(isGreen);
```

```
const greenOnly = shapes.filter((shp:Shape): boolean => {
                                  return shp.color === “green”;
                                });
```

```
const greenOnly = shapes.filter((z) => z.color === “green”);
```


Array.map(): extract all colors/num sides




```
let c: string[];

c = shapes.map((z:Shape) => {
  return z.color;
});
// OR
c = shapes.map((z) => z.color);
```

```
let sd: number[];
sd = shapes.map((w:Shape) => {
  return w.sides;
});

// OR
sd = shapes.map((w:Shape) => w.sides);
```

Array.filter() & Array.map()

Named function

```
const numbers = [2, -30, 0, 17, 9, -11];

function isPos(x:number): boolean {
  return x > 0;
}

const out = numbers.filter(isPos);
console.log(out);     // [2, 17, 19]
```

```
const numbers = [2, -30, 0, 17, 9, -11];

function plus10(x:number): number {
  return x + 10;
}

const out = numbers.map(plus10);
console.log(out); // [12, -20, 10, 27, 19, -1]
```

Fat arrow

```
const numbers = [2, -30, 0, 17, 9, -11];

const out = numbers.filter((x:number) => {
  return x > 0; });
console.log(out);     // [2, 17, 19]
```

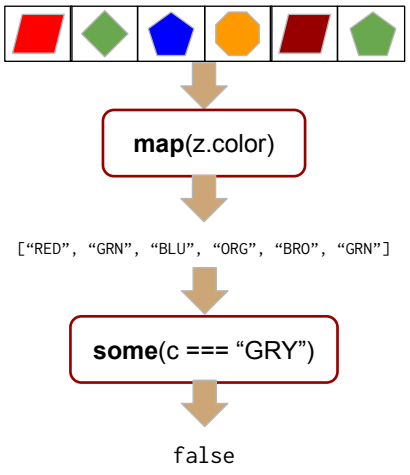
```
const numbers = [2, -30, 0, 17, 9, -11];

const out = numbers.map((x:number) => {
  return x + 10; });
console.log(out); // [12, -20, 10, 27, 19, -1]
```


Chaining Function Calls

```
// Java
String location = “Grand Rapids”;
String canyon = location.replace(“Rapids”, “Canyon”);
String caps = canyon.toUppercase();
```

```
// Java: chain the function calls
String location = “Grand Rapids”;

String grCanyon = location
  .replace(“Rapids”, “Canyon”)
  .toUppercase();
```

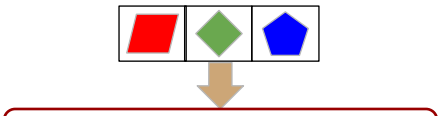
Chaining multiple Array functions




```
const shapes: Shape[];

const someGray = shapes
  .map((z:Shape) => z.color)
  .some((c:string) => c === "GRY");

console.log(someGray);
```

```
const hasGray = shapes
  .some((s:Shape) => s.color === "GRY");

console.log(someGray);
```

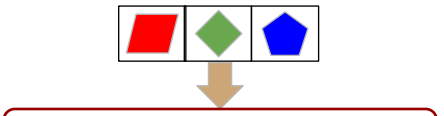

Array.flatmap(): from one to many



length = 13

```
[8, 11, 8, 11, 7, 7, 7, 7, 6.1, 6,1, 6.1, 6.1, 6.1]
```

```
let stats: number[];

stats = shapes
  .flatmap((s:Shape): number[] => {
     return s.sideDims
   });

stats = shapes
  .flatmap((s:Shape): number[] => s.sideDims);
```



length = 3

```
[[8, 11, 8, 11], [7, 7, 7, 7], [6.1, 6,1, 6.1, 6.1, 6.1]]
```

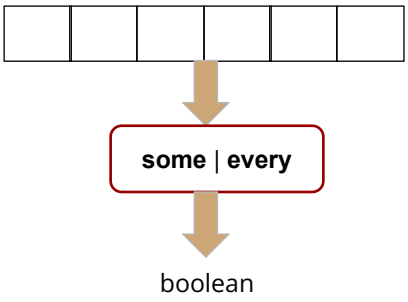
```
let stats: number[];

stats = shapes
  .map((s:Shape): number[] => {
     return s.sideDims
   });

stats = shapes
  .map((s:Shape): number[] => s.sideDims);
```

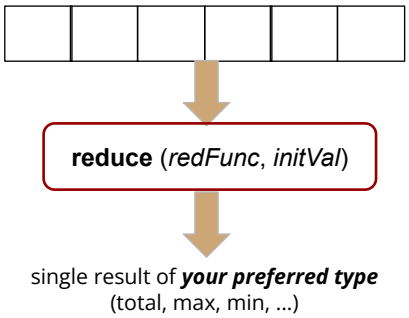
Practical Use Case of flatmap()

```
type Course = {
   name: string,
   credits: number
   classList: Array<string>
}

let allCourses: Array<Course> = [
  { name: “MTH101 Calculus”,
    credits: 4.
    classList: [/* 25 student names */]
  },
  { name: “HTM 203 Beer Brewing”,
    credits: 2,
    classList: [/* 70 student names */]
  }
]
```

```
// Find all students whose name begins with “Eli”

const studentList = allCourses
  .flatMap((c:Course) => {
    return c.classList
  })
  // you’ll get 95 names from flatMap
  .filter((who:string) => {
     return who.startsWith(“Eli”)
  });

// Or after single-return elimination
const studentList =
  allCourses
    .flatMap((c:Course) => c.classList)
    // you’ll get 95 names from flatMap
    .filter((who:string) => who.startWith(“Eli”));
```


Introducing: Array.reduce()





Not limited to only boolean output!!!

Array.reduce(): sum of values

```
const scores = [23, -31, 17, 31, 19];
const computeSum = (acc:number, curr: number): number {
   return acc + curr;
}

const totalScore = rivers.reduce(computeSum);
console.log("Total ", totalScore);  // Total 59
```

pos	acc	curr	return
1	23	-31	-8
2	-8	17	9
3	9	31	40
4	40	19	59

- *Acc is initialized from the first array item*
- *Work begins at position 1*

Array.reduce(): sum of values (with initial value)

```
const scores = [23, -31, 17, 31, 19];
const computeSum = (acc:number, curr: number): number {
   return acc + curr;
}

const totalScore = rivers.reduce(computeSum, 2000);
console.log("Total ", totalScore);  // Total 59
```

pos	acc	curr	return
0	2000	23	2023
1	2023	-31	1992
2	1992	17	2009
3	2009	31	2040
4	2040	19	2059

- *Acc is initialized from the initial value*
- *Work begins at position 0*

Array.reduce(): shortest river name

```
const rivers = ["Amazon", "Mississippi", "Nile",
                "YangTze", "Yenisei"];
const shorterOf = (acc:string, curr: string): string {
  if (curr.length < acc.length)
    return curr
  else
    return acc;
}

const riverName = rivers.reduce(shorterOf);
console.log("Shortest ", riverName);  // Nile
```

pos	acc	curr	return
1	Amazon	Mississippi	Amazon
2	Amazon	Nile	Nile
3	Nile	YangTze	Nile
4	Nile	Yenisei	Nile

- *Acc is initialized from the first array item*
- *Work begins at position 1*

Array.reduce(): shortest river name

```
const rivers = ["Amazon", "Mississippi", "Nile",
                "YangTze", "Yenisei"];

const shorterOf = (acc:string, curr: string): string {
  if (curr.length < acc.length)
    return curr
  else
    return acc;
}

// Use “”Yellow” as the initial value of riverName
const riverName = rivers.reduce(shorterOf, “Yellow”);
console.log("Shortest ", riverName);  // Nile
```

pos	acc	curr	return
0	Yellow	Amazon	Amazon
1	Amazon	Mississippi	Amazon
2	Amazon	Nile	Nile
3	Nile	YangTze	Nile
4	Nile	Yenisei	Nile

- *Acc is initialized from the provided value*
- *Work begins at position 0*

Array.reduce(): shortest name (**incorrect** initial value)

```
const rivers = ["Amazon", "Mississippi", "Nile",
                "YangTze", "Yenisei"];

const shorterOf = (acc:string, curr: string): string {
  if (curr.length < acc.length)
    return curr
  else
    return acc;
}

// Use “”Roe” as the initial value of riverName
const riverName = rivers.reduce(shorterOf, “Roe”);
console.log("Shortest ", riverName);  // Roe
```

pos	acc	curr	return
0	Roe	Amazon	Roe
1	Roe	Mississippi	Roe
2	Roe	Nile	Roe
3	Roe	YangTze	Roe
4	Roe	Yenisei	Roe

- *Acc is initialized from the provided value*
- *Work begins at position 0*

Array.reduce() with initial value

```
const rivers = ["____", ____];

const shorterLen(acc: number, curr:string):number {
  if (curr.length < acc)
    return curr.length
  else
    return acc;
}

// Use 37 to initialize riverLen
const riverLen = rivers.reduce(shorterLen, 37);

console.log("Shortest ", riverLen);  // 4
```

pos	Acc (number)	Curr (string)	Return (number)
0	37	Amazon	6
1	6	Mississippi	6
2	6	Nile	4
3	4	YangTze	4
4	4	Yenisei	4

- *Type of acc and curr may be different*
- *Type of acc and type of initial value must match*
- *Type of acc determines the type of return*

Array.reduce(): general guidelines

```
let myArray: Array<XYZ>;
```

Without initial value

```
function myFunction(prev: XYZ, curr: XYZ): XYZ {
   // More code here
   return _______;
}

const result: XYZ = myArray.reduce(myFunction);
```

With initial value

```
function myFunction(prev: resultType, curr: XYZ): resultType {
   // More code here
   return _______;
}

const initValue: resultType = _______;
const result: resultType = myArray.reduce(myFunction, initValue);
```


Reduce: Array of objects

```
type River = {
  name: string,
  countries: Array<string>,   // the river passes thru these countries
  lenInMiles: number          // river length in miles
}
```

```
const waters = Array<River> = [
  {name: “Amazon”,      countries: [“Brazil”, “Columbia”, “Peru],                    length: 4_132},
  {name: “Nile”,        countries: [“Egypt”],                                        length: 4_388},
  {name: “Mississippi”, countries: [“US”],                                           length: 2_340},
  {name: “Mekong”,      countries: [“China”, “Myanmar”, “Laos”, “Thailand”, “Vietnam”], length: 2_703},
  {name: “Ganges”,      countries: [“India”, “Bangladesh”],                          length: 1_560},
  /* more data here */
]
```

The name of the longest river?

```
type River = {
  name: string,
  countries: Array<string>,
  lenInMiles: number
}
```

```
function lengthCompare (prev: River, curr: River): River {
  if (prev.lenInMiles > curr.lenInMiles) return prev;
  else return curr;
}

let winner:River;
winner = waters.reduce(lengthCompare);
console.log(winner.name);
```

Fat arrow

```
let winner:River;

winner = waters.reduce((prev:River, curr:River): River => {
  if (prev.lenInMiles > curr.lenInMiles) return prev;
  else return curr;
});

console.log(winner.name);
```


The longest mile?

```
type River = {
  name: string,
  countries: Array<string>,
  lenInMiles: number
}
```

Option 1

```
function compLength (prev: River, curr: River): River {
  if (prev.lenInMiles > curr.lenInMiles) return prev;
  else return curr;
}

let winner:River;
winner = waters.reduce(compLength);
console.log("Longest mile is", winner.lenInMiles);
```

Option 2

```
function compRivLen (prev: number, curr: River): number {
  if (prev > curr.lenInMiles) return prev;
  else return curr.lenInMiles;
}

let winner:number;
winner = waters.reduce(compRivLen, Number.MIN_VALUE);
console.log("Longest mile is", winner);
```

Goes through most countries?

```
type River = {
  name: string,
  countries: Array<string>,
  lenInMiles: number miles
}
```

```
function countryCompare(prev: River, curr: River): River {
  if (prev.countries.length > curr.countries.length) return prev;
  else return curr;
}

let winner:River;
winner = waters.reduce(countryCompare);
console.log(winner.name);
```

Fat arrow

```
let winner:River;

winner = waters.reduce((prev:River, curr:River): River => {
  if (prev.countries.length > curr.countries.length) return prev;
  else return curr;
});

console.log(winner.name);
```


Some examples

How many green shapes?	`shapes.filter(s => s.color === “green”)` `  .length`

How many equilateral triangles?	`shapes.filter(s => s.sides === 3 &&`
	`  s.sideDims[0] === s.sideDims[1] &&`
	`  s.sideDims[1] === s.sideDims[2])`
	`  .length;`
	
	`shapes.filter(s => s.sides === 3)`
	`  .filter(s => s.sideDims[0] === s.sideDims[1] &&`
	`  s.sideDims[1] === s.sideDims[2])`
	`  .length`

Largest perimeter?	`shapes.map(shp => {`
	`  let perimeter = 0;`
	`  // Compute perimeter`
	`  return perimeter;`
	`  })`
	`  .reduce((acc:number,curr:number) => {`
	`  if (acc > curr) return acc;`
	`  else return curr;`
	`  })`

Advanced Topics:
Function Type Aliases & Generic Funcs