

---

# HIGHER-ORDER FUNCTIONS

---

# AGENDA

- ▶ Abstraktion
- ▶ Higher-Order Functions

---

# ABSTRAKTION

- ▶ En princip för bra design är DRY
- ▶ En annan är att försöka hålla saker enkla

“There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies, and the other way is to make it so complicated that there are no obvious deficiencies.”

— C.A.R. Hoare, *1980 ACM Turing Award Lecture*

---

# ABSTRAKTION

- ▶ Jämför dessa två kodsnuttar:
  - ▶ Vad gör dom? Vilken är mer lättläst? Vilken har större risker?

```
let total = 0, count = 1;
while (count <= 10) {
  total += count;
  count += 1;
}
console.log(total);
```

```
console.log(sum(range(1, 10)));
```

---

# ABSTRAKTION

- ▶ Minns ni vad vi sa om funktioner och **abstraktion**?
- ▶ Jämför följande recept

“Put 1 cup of dried peas per person into a container. Add water until the peas are well covered. Leave the peas in water for at least 12 hours. Take the peas out of the water and put them in a cooking pan. Add 4 cups of water per person. Cover the pan and keep the peas simmering for two hours. Take half an onion per person. Cut it into pieces with a knife. Add it to the peas. Take a stalk of celery per person. Cut it into pieces with a knife. Add it to the peas. Take a carrot per person. Cut it into pieces. With a knife! Add it to the peas. Cook for 10 more minutes.”

“Per person: 1 cup dried split peas, half a chopped onion, a stalk of celery, and a carrot. ¶  
Soak peas for 12 hours. Simmer for 2 hours in 4 cups of water (per person). Chop and add vegetables. Cook for 10 more minutes.”


---

# ABSTRAKTION

- ▶ I recept kan vi förlita oss på **nyckelord**
- ▶ I program kan vi förlita oss på **funktioner**
- ▶ I programmering funderar vi på att abstrahera funktionalitet när vi ser ett **mönster** (något repeteras mycket)

---

# ABSTRAKTION


- ▶ Programmering är som att bygga med lego
- ▶ Dom minsta legobitarna 
  - ▶ **values:** string, boolean, number, null....
  - ▶ **control flow:** if else, for, while, switch, ternary
- ▶ Vi bygger större legokonstruktioner
  - ▶ **functions**



---

# ABSTRAKTION

- ▶ Det är vanligt att göra något flera gånger, därför har vi **for** och **while** loopar

```
for (let i = 0; i < 10; i++) {  
  console.log(i);   
}
```

- ▶ Men for är ingen funktion, vi måste hårdkoda **vad den ska göra** och **hur många gånger** typ printa ut 10 gånger.
- ▶ Men tänk om vi vill variera detta



# ABSTRAKTION

- ▶ Vi börjar med att *variera* antalet gånger

```
function repeatLog(n) {  
  for (let i = 0; i < n; i++) {  
    console.log(i);  
  }  
}
```

- ▶ Vi abstraherar mer genom att *variera* vad den ska göra

```
function repeat(n, action) {  
  for (let i = 0; i < n; i++) {  
    action(i);  
  }  
}
```

```
repeat(3, console.log);  
// → 0  
// → 1  
// → 2
```

# ABSTRAKTION

- ▶ Vi kan skicka flera funktioner nu

```
let labels = [];  
repeat(5, i => {  
  labels.push(`Unit ${i + 1}`);  
});  
console.log(labels);  
// → ["Unit 1", "Unit 2", "Unit 3", "Unit 4", "Unit 5"]
```

- ▶ Detta är vad abstraktion betyder, vi bryr oss inte om detaljer, utan abstraherar över dessa. Vi börjar tänka på det större perspektivet

# HIGHER-ORDER FUNCTIONS

- ▶ Funktioner som tar funktioner i input eller ger funktioner i output
- ▶ Kallas "Higher-order functions"

```
function greaterThan(n) {  
  return m => m > n;  
}  
let greaterThan10 = greaterThan(10);  
console.log(greaterThan10(11));  
// → true
```

```
function noisy(f) {  
  return (...args) => {  
    console.log("calling with", args);  
    let result = f(...args);  
    console.log("called with", args, ", returned", result);  
    return result;  
  };  
}  
noisy(Math.min)(3, 2, 1);  
// → calling with [3, 2, 1]  
// → called with [3, 2, 1] , returned 1
```

# HIGHER-ORDER FUNCTIONS

- ▶ Vi kan t.o.m skapa ny flowcontrol på paritet med if/else

```
function unless(test, then) {  
  if (!test) then();  
}  
  
repeat(3, n => {  
  unless(n % 2 == 1, () => {  
    console.log(n, "is even");  
  });  
});  
// → 0 is even  
// → 2 is even
```

---

# HIGHER-ORDER FUNCTIONS

- ▶ Uppgift
  - ▶ Skapa repeat och unless, och testa

```
repeat(3, console.log);  
// → 0  
// → 1  
// → 2
```

```
repeat(3, n => {  
  unless(n % 2 == 1, () => {  
    console.log(n, "is even");  
  });  
});  
// → 0 is even  
// → 2 is even
```

---

# HIGHER-ORDER FUNCTIONS

## ▶ Uppgift

- ▶ Skapa en funktion **forEach** som tar en array, en funktion och för varje element utför funktion på varje element
  - ▶ Ex: `forEach([1,2,3], console.log) => 1, 2, 3`
- ▶ Skapa en funktion **filter** som tar en array, en funktion (test) som returnerar en boolean, och returnerar en array med de element som funktionen gett true
  - ▶ Ex: `filter([1,2,3,4], nr => nr % 2 === 0) => [2, 4]`
- ▶ Skapa en funktion **map** som tar en array, en funktion och returnerar en array med funktionen utförd på varje element
  - ▶ Ex: `map([1,2,3], nr => nr + 1) => [2,3,4]`
  - ▶ Fundera över vilka krav detta ställer på funktionen vi skickar in (purity och sidoeffekter)

# HIGHER-ORDER FUNCTIONS

## ► Reduce

- En annan vanlig sak att göra med arrayer är att "reducera" dem

```
function reduce(array, combine, start) {  
  let current = start;  
  for (let element of array) {  
    current = combine(current, element);  
  }  
  return current;  
}  
  
console.log(reduce([1, 2, 3, 4], (a, b) => a + b, 0));  
// → 10  
console.log(reduce([true, false, true, true], (a, b) => a && b, true));  
// → false  
console.log(reduce([true, true, true, true], (a, b) => a && b, true));  
// → true
```

---

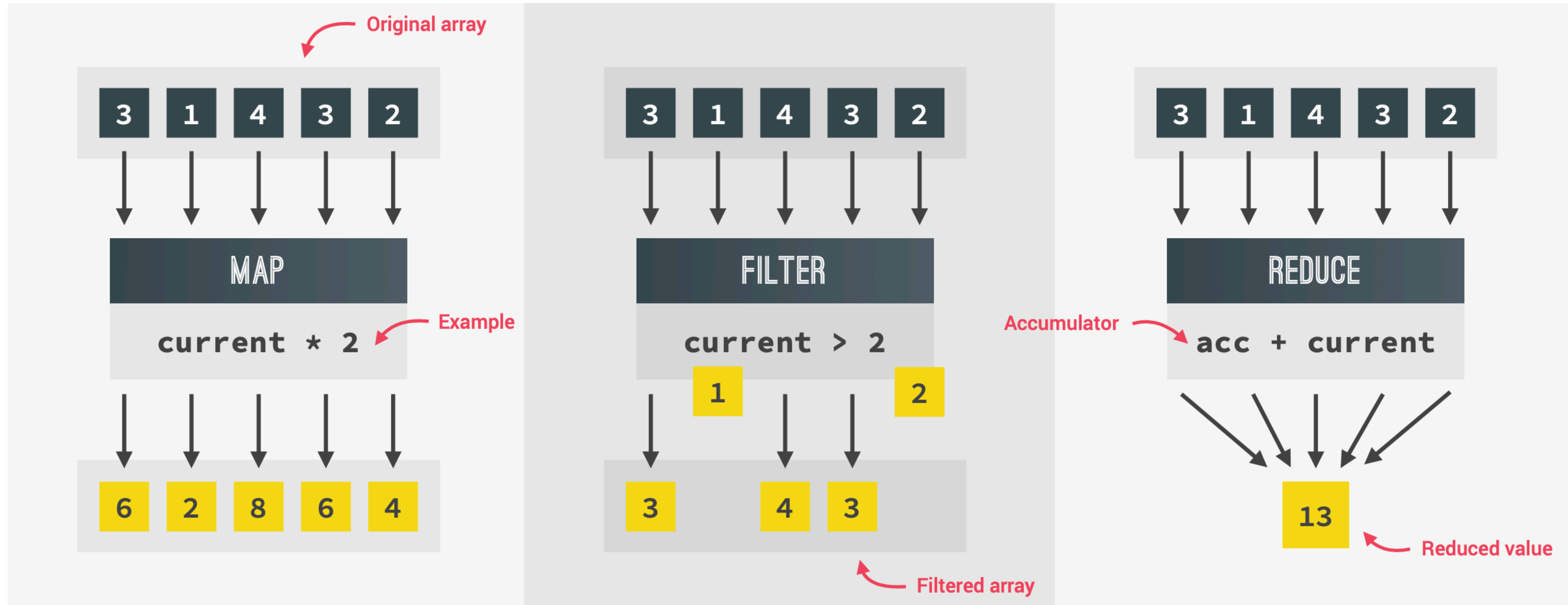
# HIGHER-ORDER FUNCTIONS

- ▶ De flesta funktioner finns faktiskt redan
  - ▶ [].forEach
  - ▶ [].filter
  - ▶ [].map
  - ▶ [].reduce
  - ▶ [].findIndex
- ▶ Uppgift: Testa dessa funktioner med godtycklig data (bevisa att du förstår)



# HIGHER-ORDER FUNCTIONS

- **map**, **filter** och **reduce** är viktiga funktioner som används ofta



---

# HIGHER-ORDER FUNCTIONS

- ▶ Uppgift
  - ▶ Implementera m.h.a reduce (sum, any, max)
    - ▶ Ex: `sum([1,2,3,4]) => 10`
    - ▶ Ex: `any([false, true, false, false]) => true`
    - ▶ Ex: `max([1,4,3,2]) => 4`

---

# HIGHER-ORDER FUNCTIONS

## ▶ Visa

- ▶ Skriv en funktion "compose" som tar 2 funktioner (a) och (b) som skapar en ny funktion där var och en av funktionerna används
  - ▶ Ex: `const add1AndLog = compose(console.log, x => x + 1)`
  - ▶ `add1AndLog(2) => loggar 3`
- ▶ För arrayer så har vi: `[], filter, sum == compose(filter, sum)`

---

# HIGHER-ORDER FUNCTIONS

- ▶ Uppgift
  - ▶ Implementera nrOfDigits m.h.a compose
    - ▶ Ex: nrOfDigits(16625) => 5
  - ▶ Tips
    - ▶ String(16625) => "16625"
    - ▶ "16625".length => 5

---

# ÖVNINGAR

- ▶ Eloquent JavaScript
  - ▶ [https://eloquentjavascript.net/05\\_higher\\_order.html](https://eloquentjavascript.net/05_higher_order.html)
    - ▶ Flattening, Your own loop, Everything

---

# LÄXA

- ▶ Eloquent JavaScript
  - ▶ [https://eloquentjavascript.net/06\\_object.html](https://eloquentjavascript.net/06_object.html)
  - ▶ T.o.m Prototypes