Klasser

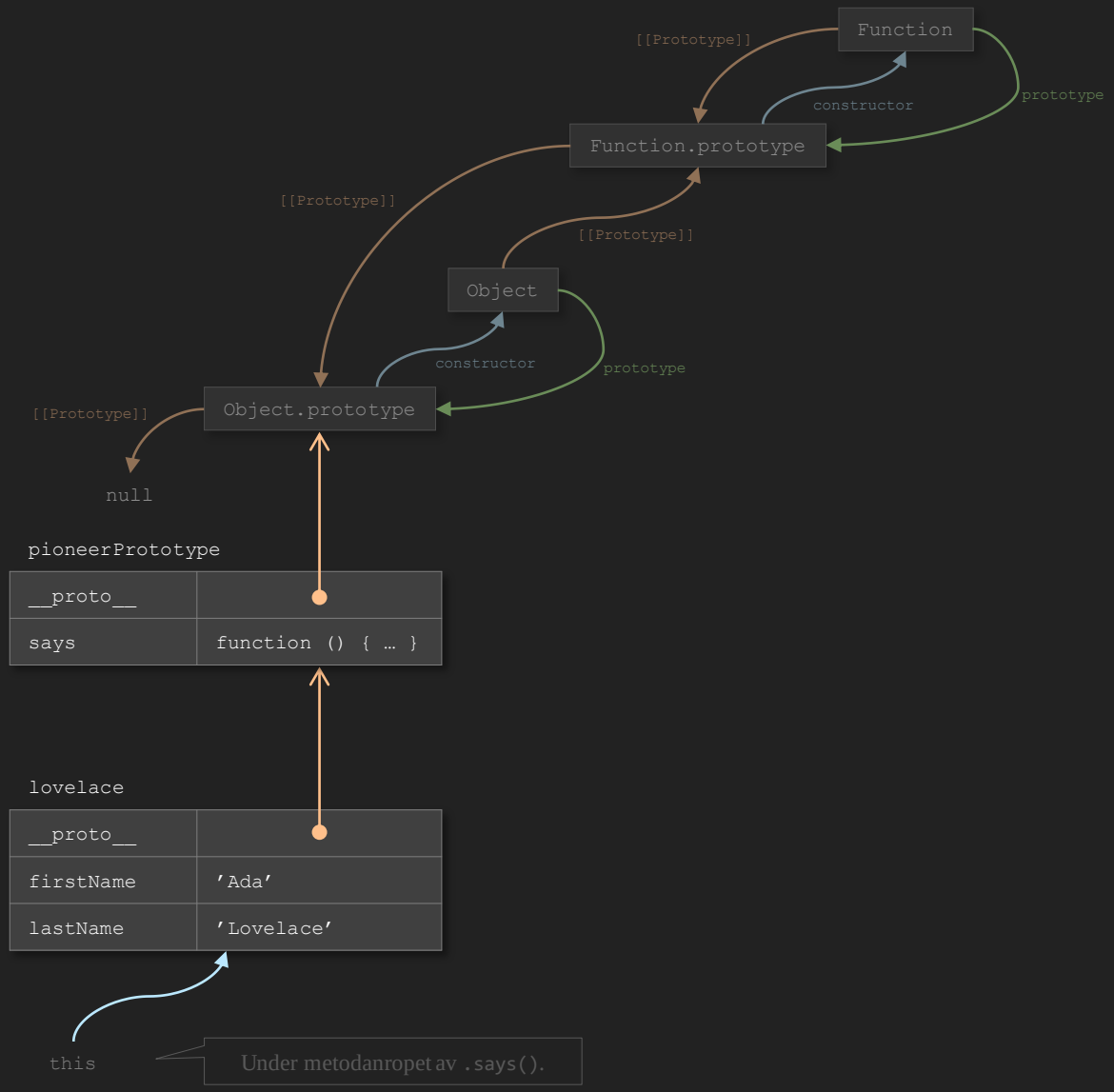
Klassdefinition som fabrik för objekt av samma typ

Allt handlar om prototypkedjor

- ✓ Alla objekt (nästan) har en kedja av ett eller flera prototypobjekt.
- ✓ Prototyper är JavaScripts viktigaste arvsmekanism.

```
const pioneerPrototype = {
  says (text) {
    return `${this.firstName} says "${text}"`
  }
}

const createPioneer = function (firstName, lastName) {
  const pioneer = Object.create(pioneerPrototype)
  pioneer.firstName = firstName
  pioneer.lastName = lastName

  return pioneer
}

const lovelace = createPioneer('Ada', 'Lovelace’)

console.log(lovelace.says('I wrote the very first algorithm!'))
```

”Factory”-designmönstret



Klassdeklaration

"JavaScript classes introduced in ECMAScript 2015 are syntactical sugar over JavaScript's existing prototype-based inheritance. The class syntax is not introducing a new object-oriented inheritance model to JavaScript. JavaScript classes provide a much simpler and clearer syntax to create objects and deal with inheritance."

från <https://developer.mozilla.org/sv-SE/docs/Web/JavaScript/Reference/Classes>

- ✓ Klass-syntax är ett kompakt sätt att sätta upp prototypkedjor.

```
class Pioneer {
  constructor (firstName, lastName) {
    this.firstName = firstName
    this.lastName = lastName
  }

  says (text) {
    return `${this.firstName} says "${text}"`
  }
}
```

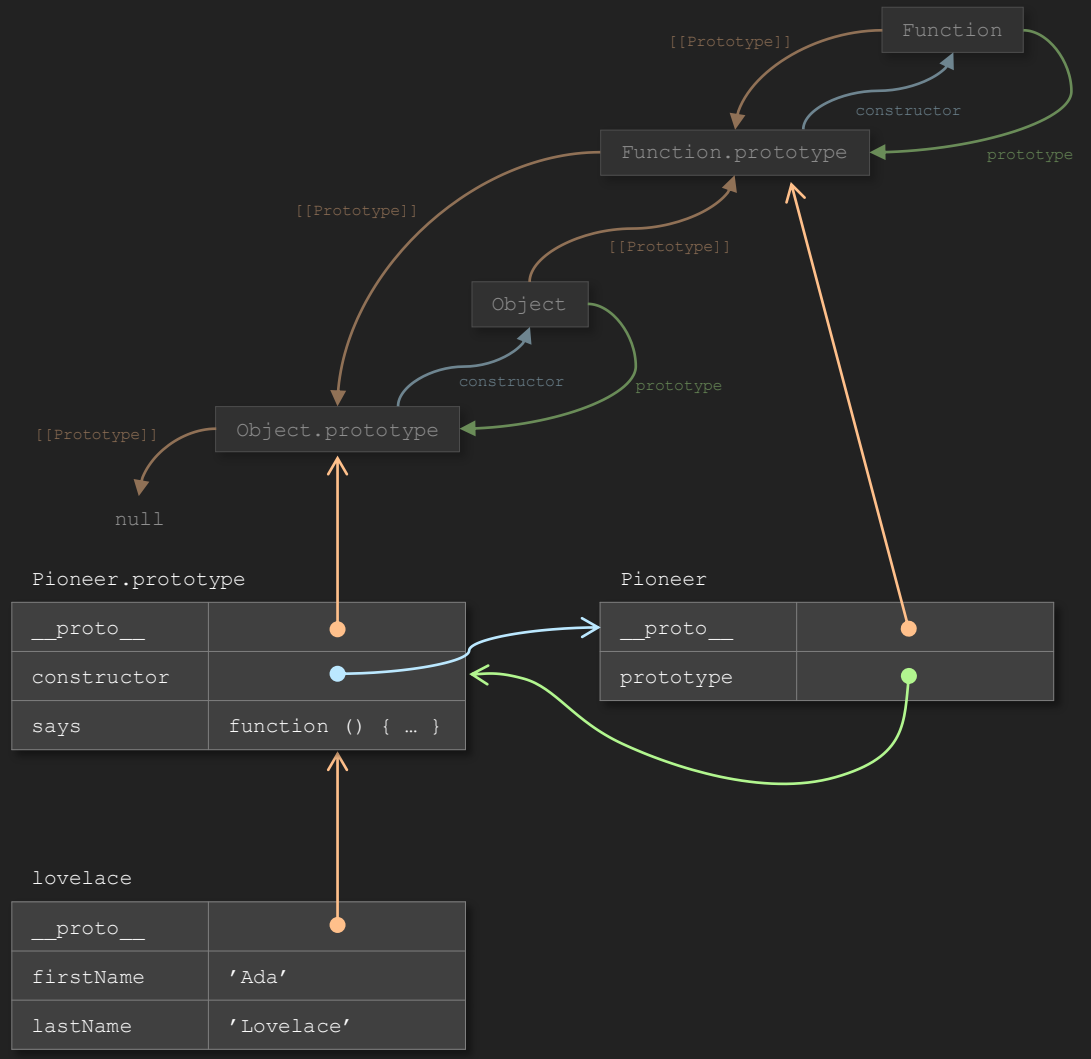
Operatorn `new` skapar ett Pioneer-objekt.

```
const lovelace = new Pioneer('Ada', 'Lovelace')

console.log(lovelace.says('I wrote the very first algorithm!'))
```

- ✓ En klass är en funktion!

```
console.log(typeof Pioneer)  // OUTPUT: function
```

- ✓ För **alla** funktioner skapas alltid egenskapen `prototype` vars värde är ett objekt innehållande egenskapen `constructor`.



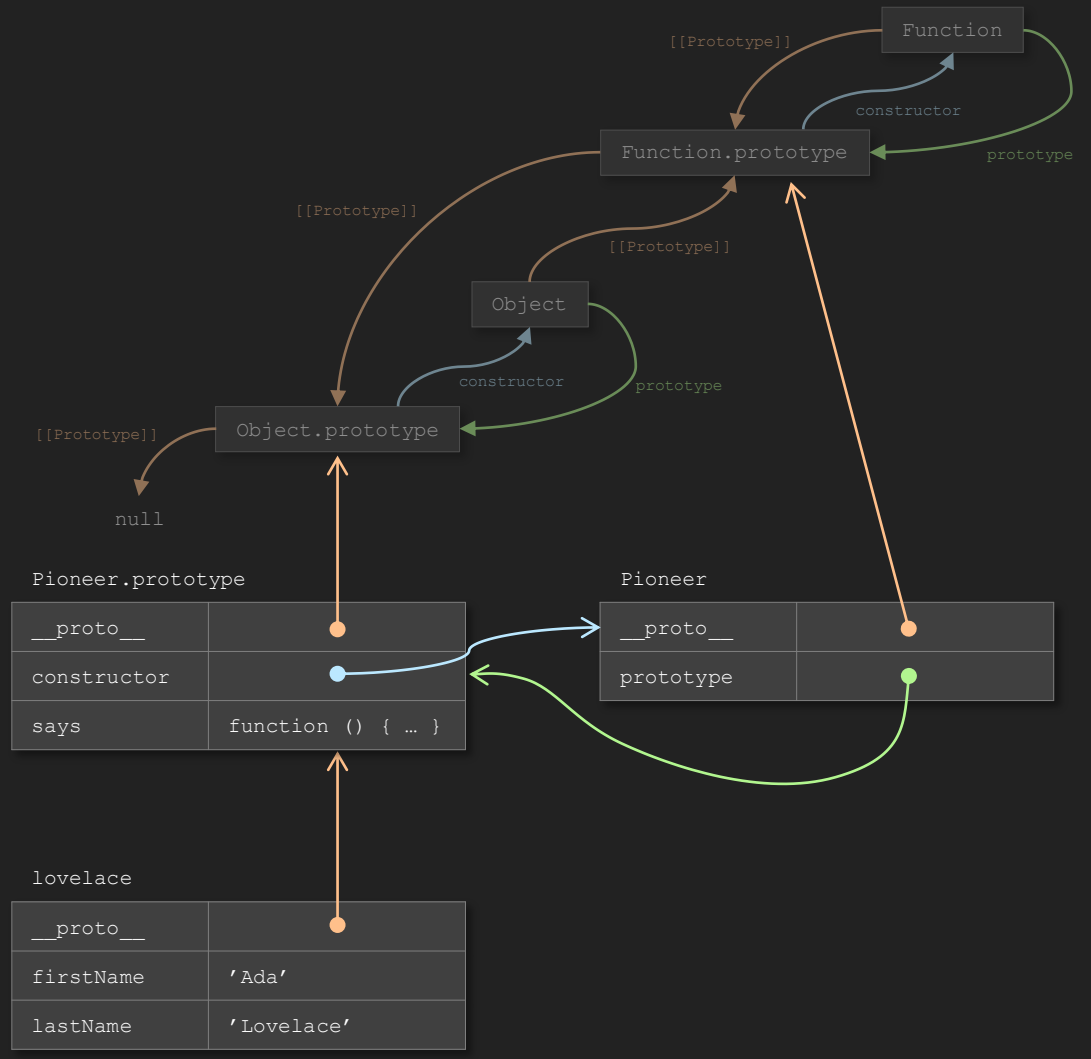
Varför klasser och inte ”*constructor/prototype*”?

- ✓ Klasser är bra att använda till exempel därför att:
  - När webbkomponenter skrivs används klasser.
  - Klasser används av många populära ramverk som React och Angular.
  - JavaScript-motorer är bra på att optimera kod som använder klasser.
  - Det är enkelt att implementera arv, även från inbyggda konstruktorfunktioner.
- ✓ De kan finnas nackdelar med klasser:
  - Skillnaden är stor mellan hur kod i klasser skrivs och vad som sedan verkligen sker.
  - Lockande att placera för mycket funktionalitet i en klass.
  - Det är lätt tillämpa arv varför det ibland görs i för stor omfattning.

```
export function Pioneer (firstName, lastName) {
  this.firstName = firstName
  this.lastName = lastName
}

Pioneer.prototype.says = function (text) {
  return `${this.firstName} says "${text}"`
}

const lovelace = new Pioneer('Ada', 'Lovelace')

console.log(lovelace.says('I wrote the very first algorithm!'))
```

Innan klasser fanns ”constructor/prototype” – ett beprövat sätt att sätta upp prototypkedjor.

Innan klasser användes (och används fortfarande) konstruktorfunktioner som fabriker för objekt.



Flera sätt att definiera en klass

- ✓ Klasser kan definieras genom klassdeklaration…

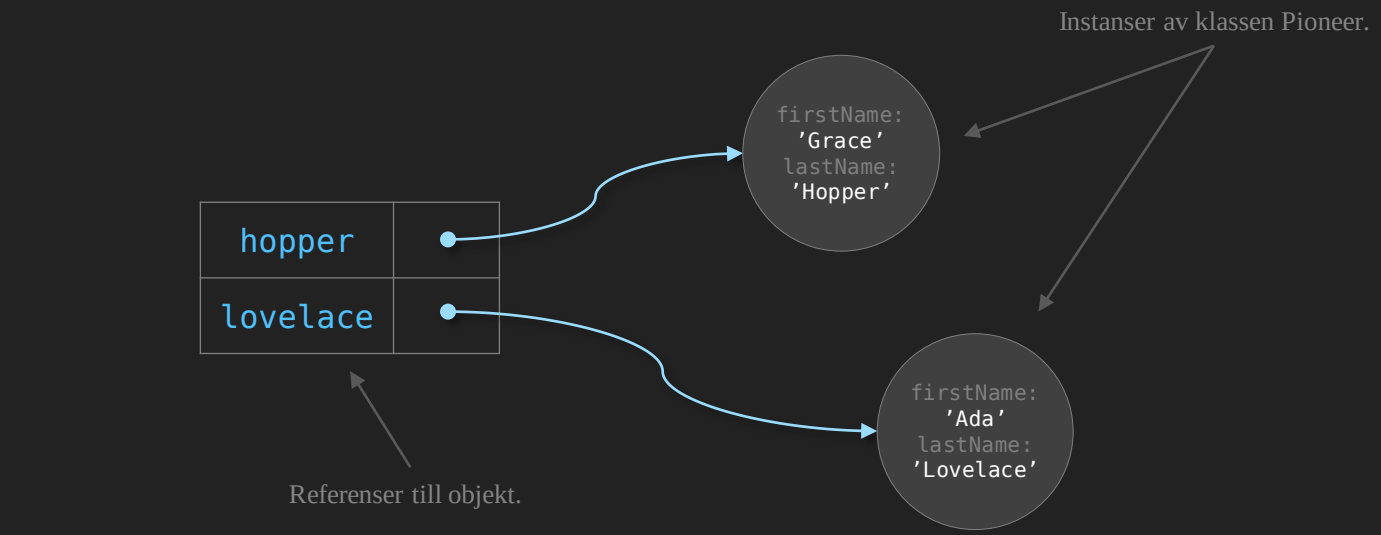
```
class Pioneer {
  constructor (firstName, lastName) {
    this.firstName = firstName
    this.lastName = lastName
  }

  says (text) {
    return `${this.firstName} says "${text}"`
  }
}
```

- ✓ …men även genom klassuttryck.

```
// Anonymous class expression
const Pioneer = class { ... }

// Named class expression
const Pioneer = class MyClass { ... }
```

- ✓ Oavsett hur en klass definieras skapas en instans (ett objekt) av, klassen genom att använda operatorn `new` följt av klassnamnet och eventuella argument.

```
const lovelace = new Pioneer('Ada', 'Lovelace')
const hopper = new Pioneer('Garce', 'Hopper')

console.log(lovelace.says('I wrote the very first algorithm!'))
console.log(hopper.says('I created the first compiler, and I found the first bug!'))
```



En klassdefinitions delar (1 av 2)

- ✓ Alla delar av klassdeklarationen nedan ger motsvarande egenskap i `Pioneer.prototype`.

```
class Pioneer {
  constructor (firstName, lastName) {
    this.firstName = firstName
    this.lastName = lastName
  }

  get fullName () {
    return `${this.firstName} ${this.lastName}`
  }

  says (text) {
    return `${this.firstName} says "${text}"`
  }
}

const lovelace = new Pioneer('Ada', 'Lovelace')

console.log(lovelace.says('I wrote the very first algorithm!')) // OUTPUT: Ada says "I wrote the very first algorithm!"

console.log(lovelace.fullName) // OUTPUT: Ada Lovelace
```

`constructor` anropas efter att den nya instansen av `Pioneer` skapats för att initiera det. Kan uteslutas om någon initiering inte behövs.

`fullName` är en ”getter”, en speciell metod som används på samma sätt som en egenskap. Denna ”getter” kapslar in dataegenskapers värden och skapar ett nytt värde utifrån dem.

`says` är en metod som har en parameter.

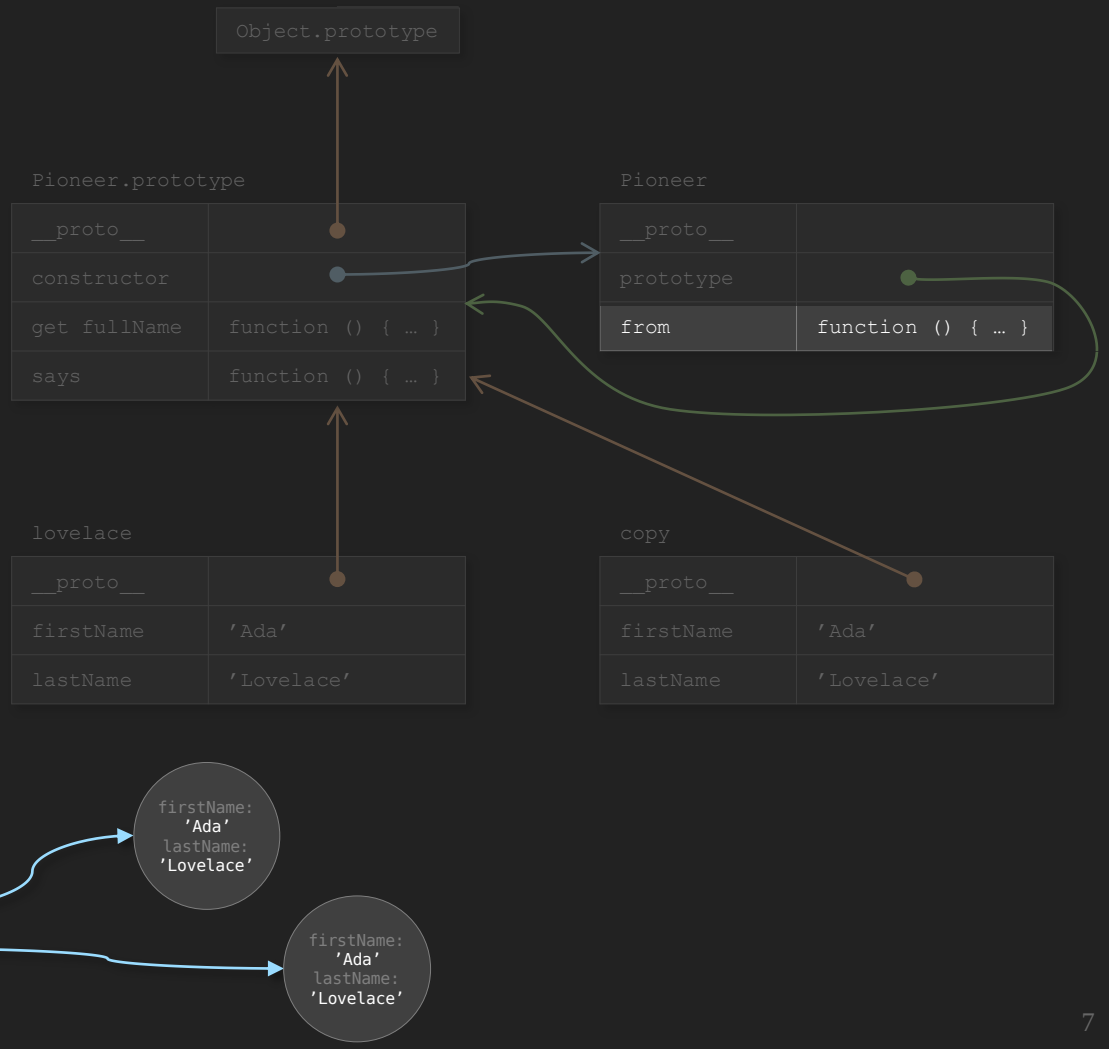
`fullName` beter sig som en vanlig egenskap, men vars värde bara kan läsas, en ”read-only”-egenskap.

En klassdefinitions delar (2 av 2)

- ✓ De delar av klassdeklarationen som föregås av `static` ger motsvarande egenskap i `Pioneer`.

```
class Pioneer {
  constructor (firstName, lastName) {
    this.firstName = firstName
    this.lastName = lastName
  }

  static from (other) {
    return new Pioneer(other.firstName, other.lastName)
  }

  get fullName () {
    return `${this.firstName} ${this.lastName}`
  }

  says (text) {
    return `${this.firstName} says "${text}"`
  }
}

const lovelace = new Pioneer('Ada', 'Lovelace')
console.log(lovelace.fullName) // OUTPUT: Ada Lovelace

const copy = Pioneer.from(lovelace)

console.log(copy.fullName) // OUTPUT: Ada Lovelace
```

`from` är en statisk metod som skapar och returnerar ett nytt `Pioneer-objekt` där värden för egenskaperna `firstName` och `lastName` hämtas från objektet som skickas som argument.



Hur kan vi skriva tydligare kod?

```
const getNumber = function () {
  return Math.floor(Math.random() * 6) + 1
}

const value = getNumber()
console.log(value) // OUTPUT: 4
```

- ✓ Genom att på något sätt kapsla in variabler och funktioner blir koden tydligare.
- ✓ Klasser kan användas till att kapsla in variabler och funktioner som hör samman.
  - En klass kan bland annat innehålla ”variabler” (egenskaper) och ”funktioner” (metoder).
- ✓ En generell klass representerande en tärning borde erbjuda en möjlighet att bestämma antalet sidor en tärning ska ha.

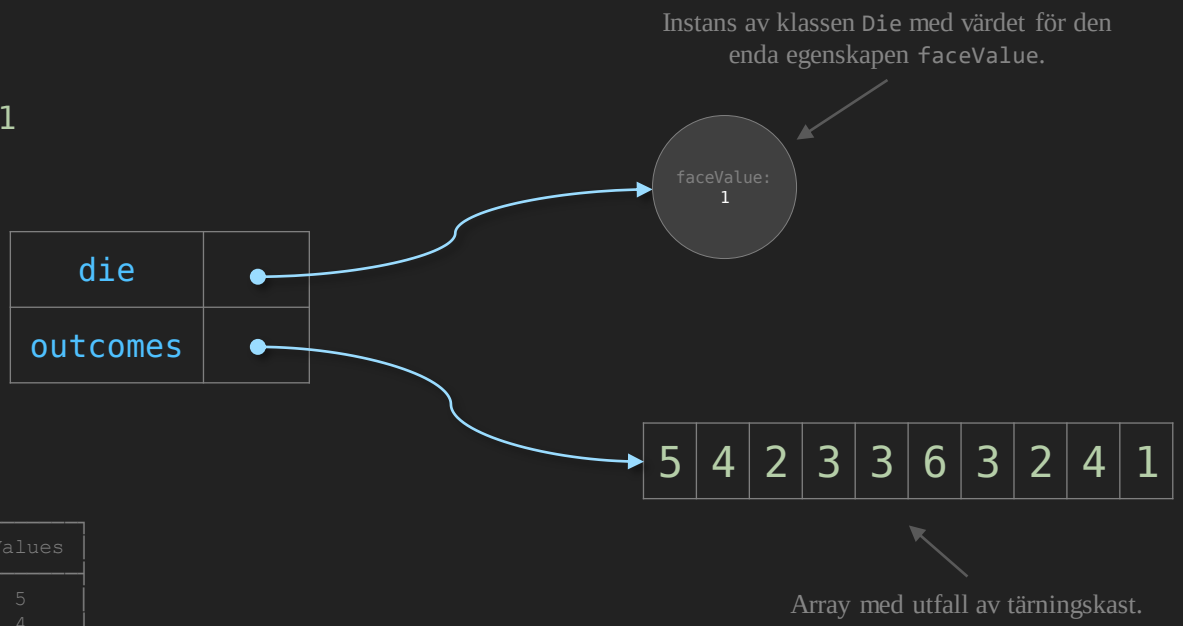
Tärningen är kastad – ett första försök

- ✓ Klassen `Die` kapslar in vad en tärning har (egenskaper) och kan (metoder).

```
class Die {
  constructor (faceValue) {
    this.faceValue = faceValue
  }

  roll () {
    this.faceValue = Math.floor(Math.random() * 6) + 1
  }
}

const die = new Die()

const outcomes = []

for (let i = 0; i < 10; i++) {
  die.roll()
  outcomes.push(die.faceValue)
}

console.table(outcomes)
```

OUTPUT:

(index)	Values
0	5
1	4
2	2
3	3
4	3
5	6
6	3
7	2
8	4
9	1



Konstruktorn initierar objektet på ett kanske inte så bra sätt. Varför?

- Bara en sexsidig tärning kan skapas.
- Behöver inte vara ett slumpat värde mellan 1 och 6.
- Kan sättas till vilken typ av värde som helst.

Tärningen är kastad – ett andra försök

- ✓ Ett slumpat värde redan från början.

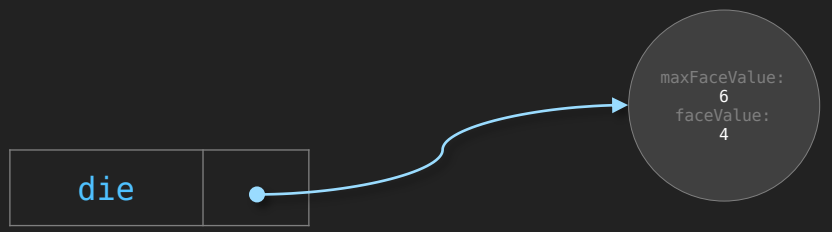
```
class Die {
  constructor (maxFaceValue = 6) {
    this.maxFaceValue = maxFaceValue
    this.roll()
  }

  roll () {
    this.faceValue = Math.floor(Math.random() * this.maxFaceValue) + 1
  }
}

const die = new Die()

console.log(die.faceValue) // OUTPUT: 4
```

Genom parametern, som har standardvärdet 6, kan antalet sidor en tärning ska ha anges. Skickas inget argument med skapas en tärning med sex sidor.

Genom att anropa metoden `roll` då objektet initieras kommer tärningens utfall vara ett värde mellan 1 och 6.

Nu slumpas ett heltal mellan 1 och det angivna maxvärdet.

Instans av klassen `Die` med värden för egenskaperna `maxFaceValue` och `faceValue` direkt efter att objektet skapats.



Det finns fortfarande problem som kommande presentation får ta en titt på!

- Egenskaperna kan fortfarande sättas till vilket värde som helst! Går det att ”skydda” egenskapers värden? Ja!