Object georiënteerd programmeren

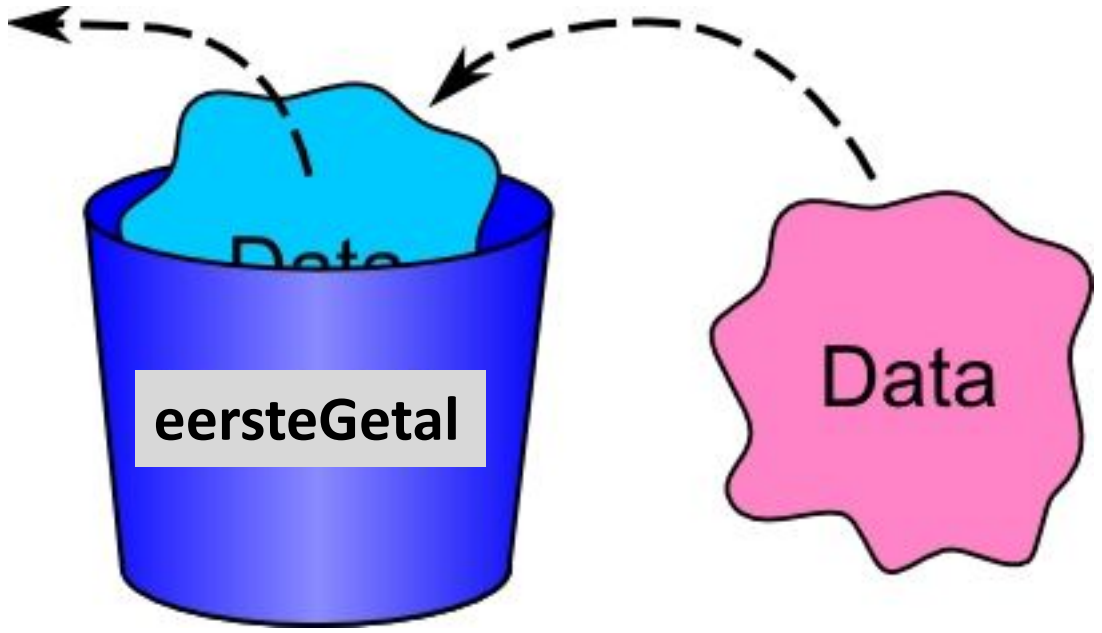
Even opfrissen…

JavaScript

Wat zijn variabelen?

- Een variabele is een plek in het geheugen waar je een gegeven in kunt zetten. Deze plek geef je een naam in de programmeertaal die je gebruikt.

- Denk aan een bakje (=geheugenplek) met een etiket (=naam) erop, waar je een getal, tekst etc. (=gegeven) in kunt stoppen.



Variabelen en datatypen

- Javascript bepaalt tijdens het uitvoeren welk soort informatie in een variabele wordt bewaart. Dit heet het datatype

```
var i = 0;
```

number

```
var gameOver = true;
```

boolean

```
var startTekst = "Welkom bij de game";
```

string

Variabelen en datatypen

- Javascript bepaalt tijdens het uitvoeren welk soort informatie in een variabele wordt bewaart. Dit heet het datatype

```
let i = 0;

let gameOver = true;

let startTekst = "Welkom bij de game";
```

je mag ook ‘let’ gebruiken in plaats van ‘var’. Dat is niet verplicht maar kan je helpen bij het oplossen en voorkomen van fouten

number

boolean

string

If-statement

- met ‘if’ kun je een programma code wel of niet laten uitvoeren, afhankelijk van de situatie

```
if (mouseX === 0) {
  println(”muis zit helemaal links”);
}
else {
  println(”muis zit niet helemaal links”);
}
```

Herhalingen

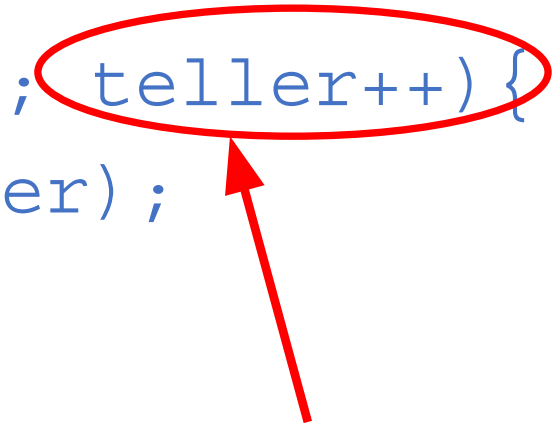
- met een for-loop laat je code herhalen, bijvoorbeeld voor elk element in een lijst of een vast aantal keren.

```
for (let teller=0; teller < 10; teller++){
  println(”teller is: ” + teller);
}
```

Herhalingen

- met een for-loop laat je code herhalen, bijvoorbeeld voor elk element in een lijst of een vast aantal keren.

```
for (let teller=0; teller < 10; teller++){
  println(”teller is: ” + teller);
}
```



Herhalingen

- met ‘for’ kun je een programma code laten herhalen, afhankelijk van de situatie

```
for (let teller=0; teller < 10; teller++){
  println(”teller is: ” + teller);
}
```

- Wat verschijnt er op het scherm?

Herhalingen

- Naast ‘for’ is er in JavaScript nog een andere structuur waarmee je voor herhaling kunt zorgen. Welke?

Herhalingen

- Naast ‘for’ is er in JavaScript nog een andere structuur waarmee je voor herhaling kunt zorgen. Welke?

```
while
```

Herhalingen

- Naast ‘for’ is er in JavaScript nog een andere structuur waarmee je voor herhaling kunt zorgen. Welke?

```
let teller = 0;
while (teller < 10) {
  println(”teller is: ” + teller);
  teller++;
}
```

Arrays

Een **array** is een rij van variabelen in een programmeertaal.

- Het volgnummer in de rij (dit heet de index) wordt gebruikt om een losse variabele (dit heet een element van de array) aan te wijzen.

Arrays

Voorbeeld:

```
// maak array met strings en print elementen

let namen = [“achmed”,”bert”,”carla”];

print(namen[0]); // drukt de de naam achmed af

print(namen[1]); // drukt de de naam bert af

print(namen[2]); // drukt de de naam carla af
```

Arrays & herhaling is een krachtig duo

Voorbeeld:

```
// maak array met strings en print elementen

let namen = [“achmed”,”bert”,”carla”,”indy”,”oxana”];

for (let i=0; i<namen.length; i++) {

  print(namen[i]); // drukt de de naam op index i af

}
```

Wat is een functie?

- Een functie is een stuk programma met een door de programmeur gekozen naam.
- Een functie is een stuk programma dat je vaker kunt gebruiken.
- Een functie is een samenhangend stuk van een programma.
- Een functie is een stuk van een programma dat je later eenvoudig kunt aanpassen of uitbreiden.

Alle bovenstaande uitspraken zijn waar.

Teken gezicht

```
1  var x = 50;
2  ellipse(x, 100, 50, 50); //hoofd
3  ellipse(x-10, 90, 10, 10); //oog links
4  ellipse(x+10, 90, 10, 10); //oog rechts
5  ellipse(x, 110, 30, 10); // mond
6
7
8
```

Teken 2 gezichten

```
1  var x = 50;
2  ellipse(x, 100, 50, 50); //hoofd
3  ellipse(x-10, 90, 10, 10); //oog links
4  ellipse(x+10, 90, 10, 10); //oog rechts
5  ellipse(x, 110, 30, 10); // mond
6
7  x = 110;
8  ellipse(x, 100, 50, 50); //hoofd
9  ellipse(x-10, 90, 10, 10); //oog links
10 ellipse(x+10, 90, 10, 10); //oog rechts
11 ellipse(x, 110, 30, 10); // mond
12
```

Teken 2 gezichten, zonder dubbele code

```
1  var tekenSmiley = function(x) {
2      ellipse(x, 100, 50, 50); //hoofd
3      ellipse(x-10, 90, 10, 10); //oog links
4      ellipse(x+10, 90, 10, 10); //oog rechts
5      ellipse(x, 110, 30, 10); // mond
6  };
7
8  tekenSmiley(50);
9  tekenSmiley(110);
10
```

Liever:

```
function tekenSmiley(x) {
  ellipse(x, 100, 50, 50);
  ellipse(x-10, 90, 10, 10);
  ellipse(x+10, 90, 10, 10);
  ellipse(x, 110, 30, 10);
}
```

OOP les 1

Objecten, klassen attributen en methoden

Variabelen en datatypen

- Javascript bepaalt tijdens het uitvoeren welk soort informatie in een variabele wordt bewaart. Dit heet het datatype

```
var i = 0;

var gameOver = true;

var startTekst = "Welkom bij de game";
```

number

boolean

string

Logische eenheden

- In je code heb je vaak stukjes informatie en / of functionaliteit die bij elkaar horen.
- bijvoorbeeld:
  - de x- en y- waarde van een bal, evt. met horizontale en verticale snelheden, samen met de code om de bal een stukje te verplaatsen.
  - evenzo van een kogel, auto, speler, enz enz
  - de positie, titel en grootte van een 'knop', met daarbij de code die uitgevoerd wordt als je op de knop klikt

Logische eenheden

- JavaScript kent daarvoor objecten.
- Verzameling van waarden met een label

```
var knopA = { x : 30,
              y : 100,
              breedte : 200,
              hoogte : 50,
              titel : “Niet klikken”
            };
rect(knopA.x, knopA.y, knopA.breedte, knopA.hoogte);
text(knopA.titel, knopA.x + 10, knopA.y + 10, knopA.breedte-20, knopA.hoogte-20);
```

Logische eenheden

- Of bij een spelletje met een vallende appel

```
var appel = { x: 300,
              y: 600,
              speed: 3
            }
```

- En dan verderop:

```
appel.y = appel.y + appel.speed;
```

Logische eenheden

- Of, in het geval van onze simulator:

```
var appel = { x: 300,
              y: 600,
              speed: 3
            }
```

- En dan verderop:

```
appel.y = appel.y + appel.speed;
```

- Wat te doen bij meerdere appels? appelA, appelB, appelC?

Logische eenheden

- Nog beter: als naamloze objecten in een array

```
var appels = [ { x: 300,
                 y: 600,
                 speed: 3
               },
               { x: 800,
                 y: 300,
                 speed: 1
               } // etcetera
             ];
```

```
for (var i=0; i < appels.length; i++) {
  appels[i].y = appels[i].y + appels[i].speed;
}
```

Logische eenheden

- De updatecode hoort eigenlijk ook bij het object. Dat doe je zo:

```
var appel = { x: 300,
              y: 600,
              speed: 3,
              update() {
                this.y = this.y + this.speed;
              }
            }
appel.update();
```

- Waarom ‘this’?
- -> De code in update kan niet ‘weten’ dat het object beschikbaar is onder het label ‘appel;

Wat is wat?

```
var appel = { x: 300,
              y: 600,
              speed: 3,
              update() {
                this.y = this.y + this.speed;
              }
appel.update();
```

attributen

methode

Probleempje…

```
var appels = [ { x: 300,
                 y: 600,
                 speedX: 2,
                 speedY: -3,
                 update() {
                   this.y = this.y + this.speed;
                 }
               },
               { x: 800,
                 y: 300,
                 speedX: -4,
                 speedY: 1,
                 update() {
                   this.y = this.y + this.speed;
                 }
               } // etcetera
             ];
```

```
for (var i=0; i<appels.length; i++) {
  appels[i].update()
}
```

Dubbele methoden

- Voor ieder object opnieuw de methodes schrijven is zonde van de tijd en opslagruimte.
- Waarom kunnen we geen objecten maken van eerder gemaakt ontwerp?
- Dat kan met de beschrijving van een klasse:

Beschrijf de class Appel

```
class Appel {
    x;
    y;
    speed;

    constructor(x, y, speed) {
        this.x = x;
        this.y = y;
        this.speed = speed
    }
}
```

Beschrijf de class Appel (met update-methode)

```
class Appel {
   x;
   y;
   speed;

   constructor(x, y, speed) {
      this.x = x;
      this.y = y;
      this.speed = speed;
   }

  update() {
    this.y = this.y + this.speed;
  }
}
```

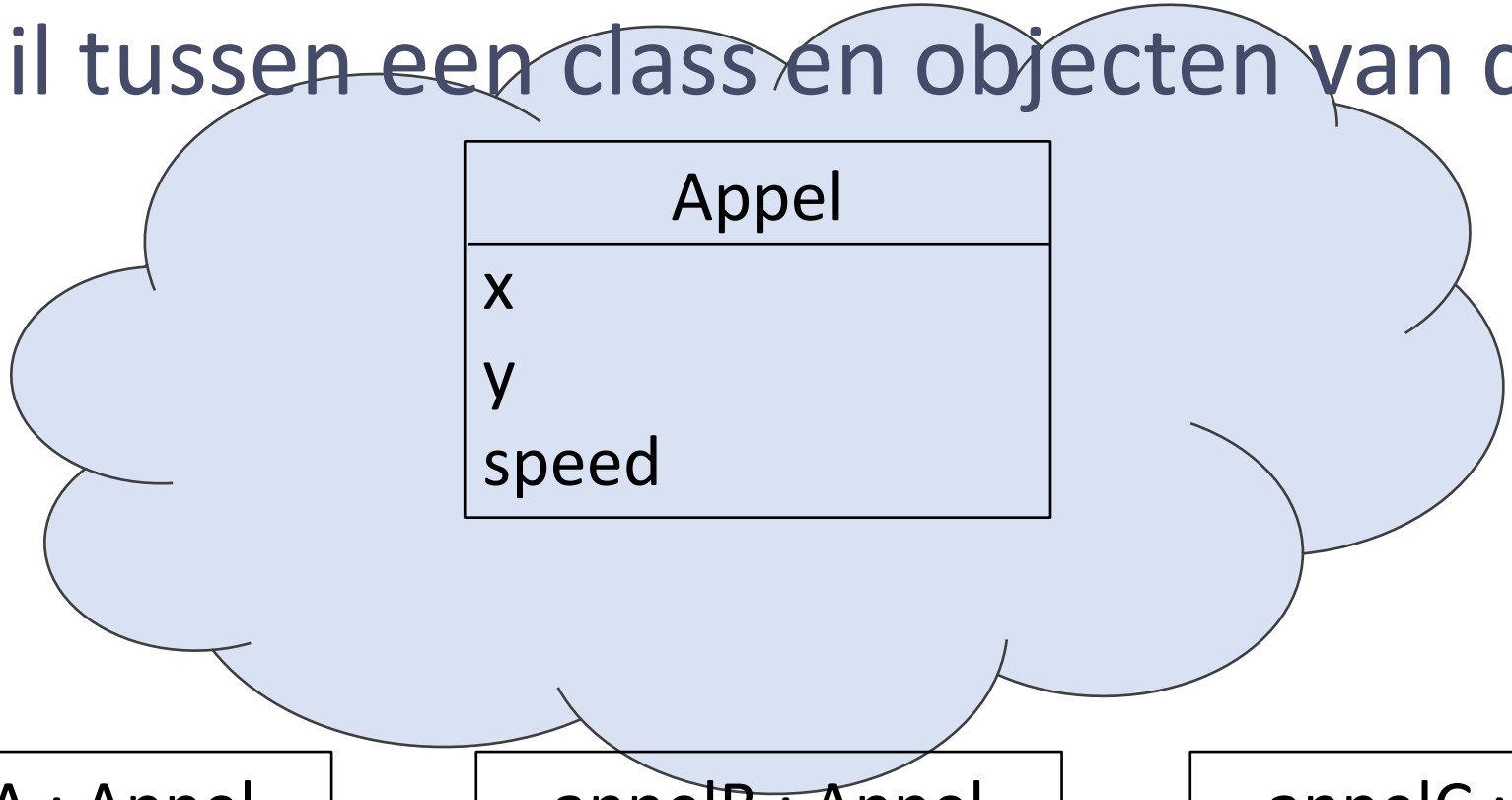
Beschrijf de class Mens (nu met deel van update)

```
class Appel {
   x;
   y;
   speed;

   constructor(x, y, speed) {
      this.x = x;
      this.y = y;
      this.speed = speed;
   }

  update() {
    this.y = this.y + this.speed;
  }
}
```

constructor wordt aangeroepen met 'new', zoals:

```
var appel = new Appel(50, 50, -7);
```

Verschil tussen een class en objecten van die class:



appelA : Appel	
x	= 50
y	= 50
speed	= 2

appelB : Appel	
x	= 74
y	= 24
speed	= 4

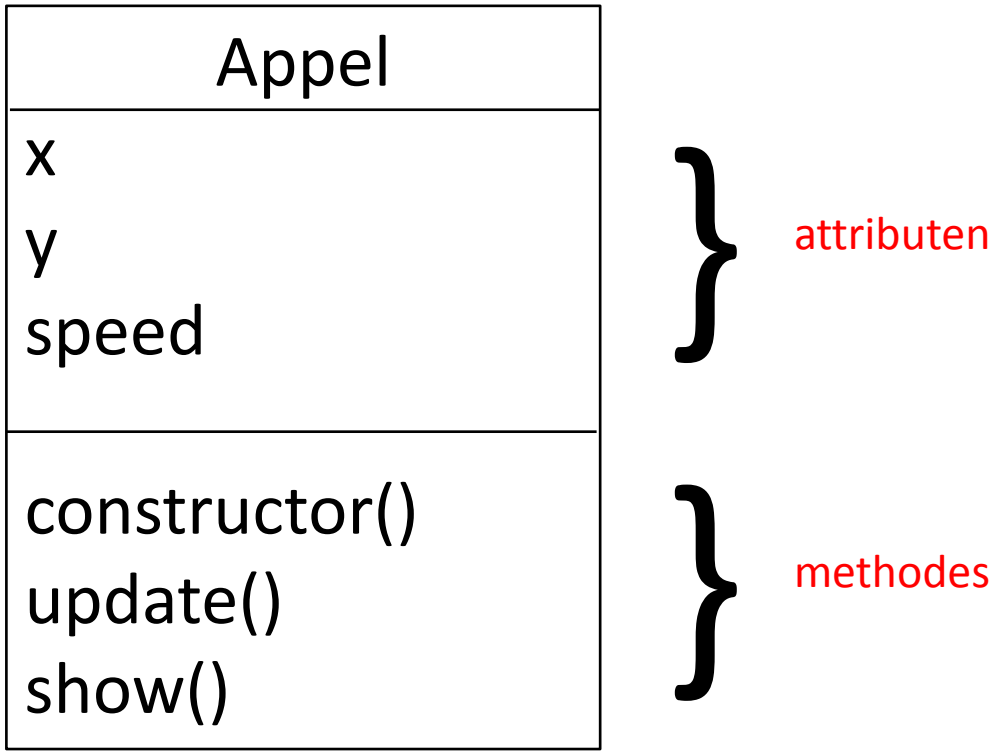
appelC : Appel	
x	= 150
y	= 91
speed	= 5

Hoe definieer ik een class?

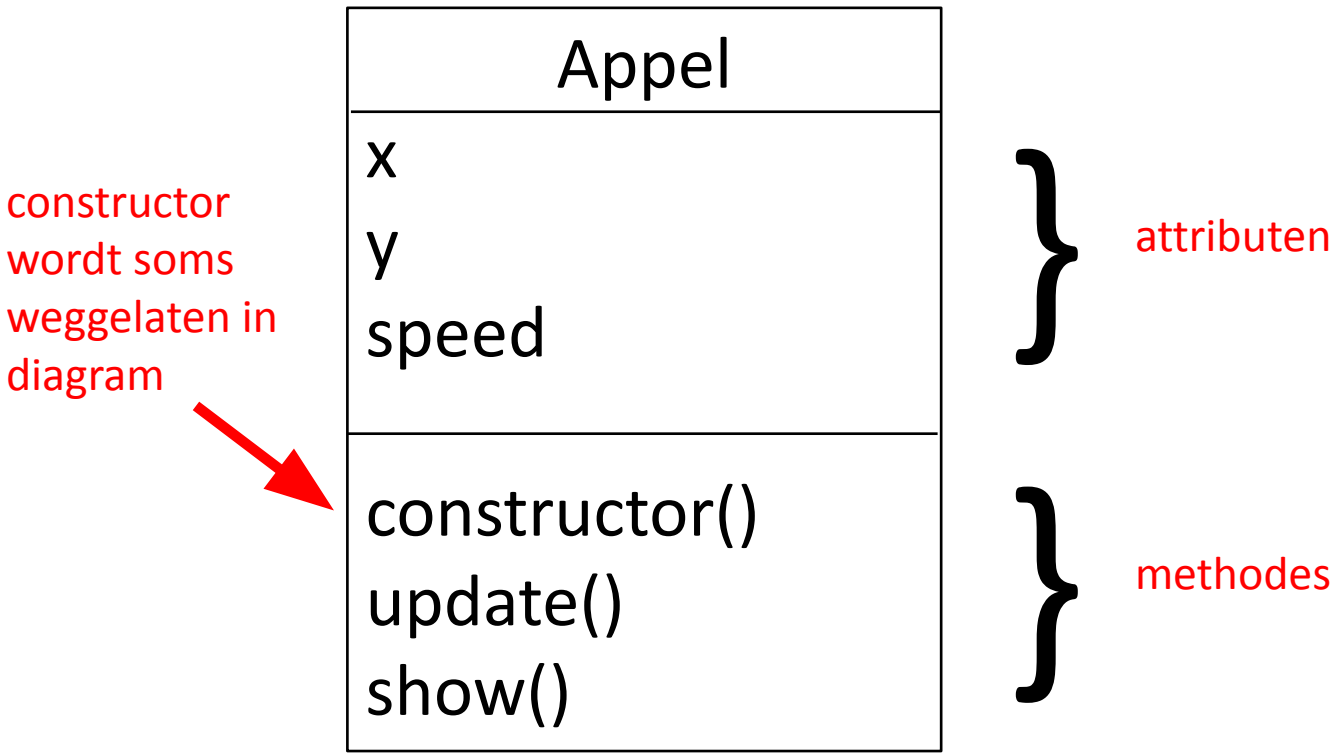
```
class <Naam van de class> {
  attribuut1;
  attribuut2;

  constructor(parameter1, parameter2) {
    this.attribuut1 = parameter waarvan je de waarde wilt gebruiken;
    this.attribuut2 = parameter waarvan je de waarde wilt gebruiken;
  }

  methodenaam() {
    // code die uitgevoerd moet worden

    return <waarde>; // alleen als er een waarde teruggegeven moet worden
  }
}
```

```
class Bal {
  x;
  y;
  speedX;
  speedY;

  constructor(x, y, speedX, speedY) {
    this.x = x;
    this.y = y;
    this.speedX = speedX;
    this.speedY = speedY;
  }

  show() {
    fill(255, 100, 255);
    ellipse(this.x, this.y, 80, 80);
  }

  update() {
    this.x = this.x + this.speedX;
    this.y = this.y + this.speedY;

    // hier moet ook de code voor het kaatsen komen
    // . . .
  }
```

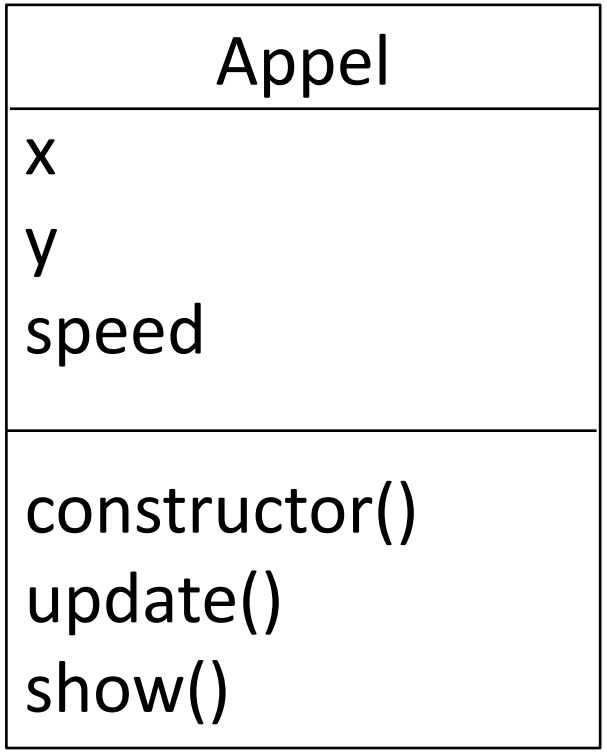
Een diagram praat vaak makkelijker



Een diagram praat vaak makkelijker



Een diagram praat vaak makkelijker



Dit is een klassendiagram

Objectdiagram

Soms wil je de toestand van objecten in een diagram zetten. Dat doe je zo:

appelA : Appel
x = 50
y = 50
speed = 2

appelB : Appel
x = 74
y = 24
speed = 4

appelC : Appel
x = 150
y = 91
speed = 5

Objectdiagram

Soms wil je de toestand van objecten in een diagram zetten. Dat doe je zo:

objectnaam : klassenaam

appelA : Appel
x = 50
y = 50
speed = 2

appelB : Appel
x = 74
y = 24
speed = 4

appelC : Appel
x = 150
y = 91
speed = 5

in een objectdiagram zet je geen methodes

Superclasses en overerving

Stel…

Appel
x y speed punten
constructor() update() show()

- Appels hebben punten (bijv. +1)

Stel…

Appel
x y speed points
constructor() update() show()

RotteAppel
x y speed points
constructor() update() show()

- Appels hebben punten (bijv. +1)
- Er komt af en toe een rotte appel voorbij. Deze heeft een minpunten (bijv. -1) en ziet er anders uit.
- Waar zitten de verschillen?

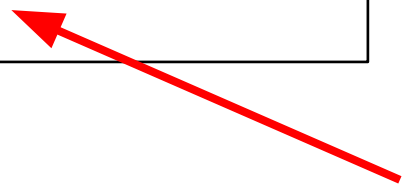
Stel…

Appel
x y speed points
constructor() update() show()

RotteAppel
x y speed points
constructor() update() show()

- Appels hebben punten (bijv. +1)
- Er komt af en toe een rotte appel voorbij. Deze heeft een minpunten (bijv. -1) en ziet er anders uit.

- Waar zitten de verschillen?



Een rotte appel wordt anders getekend.
Wordt geregeld in show().

show() :

Appel

```
show() {
    noStroke();
    fill(“red”);
    rect(this.x, this.y, 20, 20);
}
```

RotteAppel

```
show() {
    noStroke();
    fill(“brown”);
    rect(this.x, this.y, 20, 20);
}
```

Stel…

Appel
x y speed points
constructor() update() show()

RotteAppel
x y speed points
constructor() update() show()

- Denk na: als een gewone appel altijd 1 punt is en een rotte appel altijd -1 punt, hoe zou je dat dan programmeren?

Mogelijke constructor:

Appel

```
constructor(x, y, speed, points) {
    this.x = x;
    this.y = y;
    this.speed = speed;
    this.points = points;
}
```

RotteAppel

```
constructor(x, y, speed, points) {
    this.x = x;
    this.y = y;
    this.speed = speed;
    this.points = points;
}
```

Mogelijke constructor:

Appel

```
constructor(x, y, speed, points) {
    this.x = x;
    this.y = y;
    this.speed = speed;
    this.points = points;
}
```

MAAR: dat betekent dat je altijd een dezelfde waarde (1) voor de punten aan de constructor meegeeft:

```
var appelA = new Appel(200, -100, 3, 1);
var appelB = new Appel(400, -250, 2, 1);
var appelC = new Appel(600, -200, 6, 1);
```

Betere constructor:

Appel

```
constructor(x, y, speed) {
    this.x = x;
    this.y = y;
    this.speed = speed;
    this.points = 1;
}
```

RotteAppel

```
constructor(x, y, speed) {
    this.x = x;
    this.y = y;
    this.speed = speed;
    this.points = -1;
}
```

```
var appelA = new Appel(200, -100, 3);
var rotteAppelA = new RotteAppel(400, -250, 2);
var appelB = new Appel(600, -200, 6);
```

Betere constructor:

Appel

```
constructor(x, y, speed) {
    this.x = x;
    this.y = y;
    this.speed = speed;
    this.points = 1;
}
```

RotteAppel

```
constructor(x, y, speed) {
    this.x = x;
    this.y = y;
    this.speed = speed;
    this.points = -1;
}
```

```
var appelA = new Appel(200, -100, 3);
var rotteAppelA = new RotteAppel(400, -250, 2);
var appelB = new Appel(600, -200, 6);
```

De afspraken over de punten van de appels zit nu in de classes. Veel ongevoeliger voor fouten

Stel…

Appel
x y speed points
constructor() update() show()

RotteAppel
x y speed points
constructor() update() show()

- Denk na: als een gewone appel altijd 1 punt is en een rotte appel altijd -1 punt, hoe zou je dat dan programmeren?

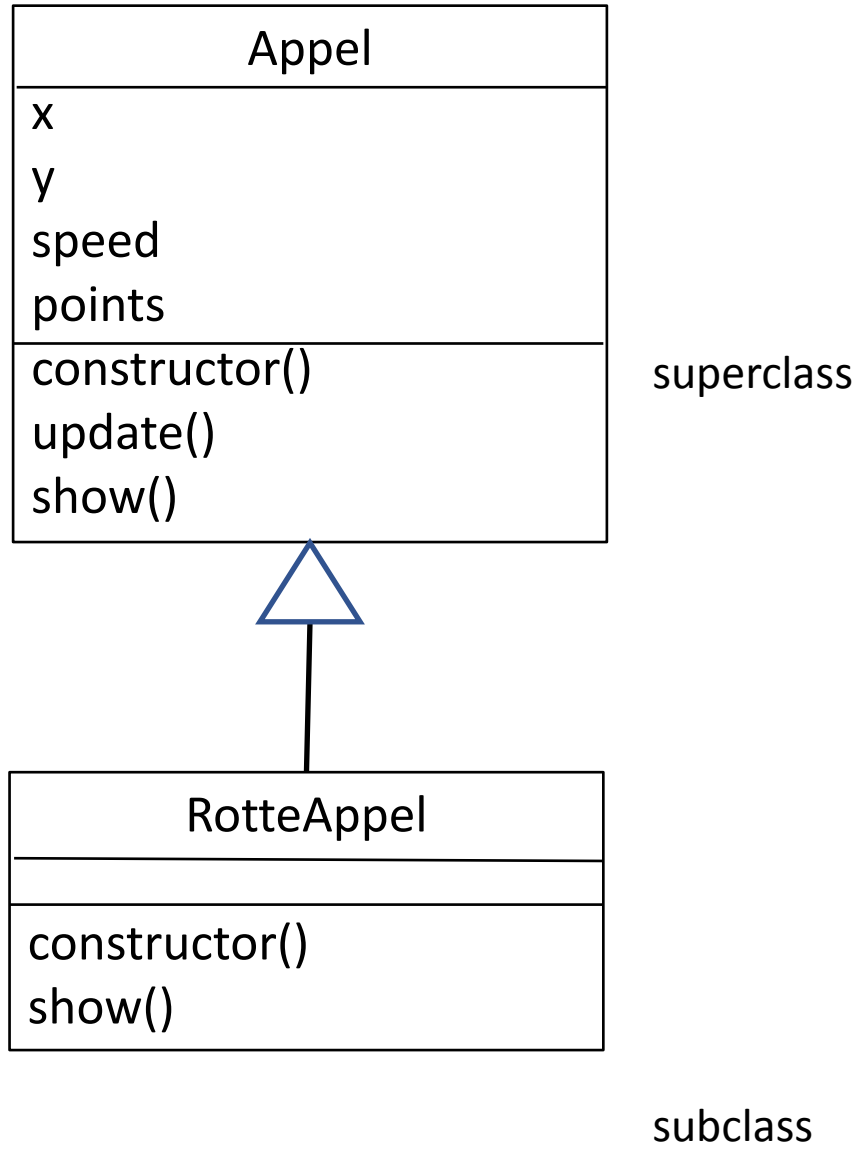
Dubbele code

- De klasse Appel heeft exact dezelfde attributen en methoden als de klasse RotteAppel, maar de methode show en de constructor zijn (een beetje) anders.
- In logisch opzicht is dit ook zo: een rotte appel is een ‘speciaal soort’ appel.
- Dubbele code is teveel werk
- Dubbele code is foutgevoelig

subclassing

Subclassing

- De klasse RotteAppel erft alle attributen en methoden van Mens
- De klasse RotteAppel heeft een eigen implementatie van de methode show(), en een eigen constructor



Subclassing in code – show()

```
class RotteAppel extends Appel {
// constructor komt nog

show() {
    noStroke();
    fill(“red”);
    rect(this.x, this.y, 20, 20);
    fill(“brown”);
    ellipse(this.x+10, this.y+10, 10, 10)
  }
}
```

- Het keyword extends geeft aan dat een klasse een subklasse is
- RotteAppel heeft alle attributen en methoden van Appel.
- show() van RotteAppel ‘overschaduwd’ show() van Appel

Subclassing in code – show()

```
class RotteAppel extends Appel {
  constructor(x, y, speed) {
    this.x = x;
    this.y = y;
    this.speed = speed;
    this.points = -1;
  }

  show() {
    noStroke();
    fill(“red”);
    rect(this.x, this.y, 20, 20);
    fill(“brown”);
    ellipse(this.x+10, this.y+10, 10, 10)
  }
}
```

```
class Appel {
  x;
  y;
  speed;
  points;

  constructor(x, y, speed) {
    this.x = x;
    this.y = y;
    this.speed = speed;
    this.points = 1;
  }

  show() {
    noStroke();
    fill(“red”);
    rect(this.x, this.y, 20, 20);
  }
}
```

Subclassing in code – show()

```
class RotteAppel extends Appel {
  constructor(x, y, speed) {
    this.x = x;
    this.y = y;
    this.speed = speed;
    this.points = -1;
  }

  show() {
    super.show();
    fill(“brown”);
    ellipse(this.x+10, this.y+10, 10, 10)
  }
}
```

Subclassing in code – constructor

```
class RotteAppel extends Appel {
  constructor(x, y, speed) {
    this.x = x;
    this.y = y;
    this.speed = speed;
    this.points = -1;
  }

  show() {
    super.show();
    fill(“brown”);
    ellipse(this.x+10, this.y+10, 10, 10)
  }
}
```

- De eerste drie regels van de constructor van RotteAppel komen letterlijk overeen met de constructor van Appel.

Subclassing in code – constructor

```
class RotteAppel extends Appel {
  constructor(x, y, speed) {
    this.x = x;
    this.y = y;
    this.speed = speed;
    this.points = -1;
  }

  show() {
    super.show();
    fill(“brown”);
    rect(this.x, this.y, 20, 20);
  }
}
```

```
class Appel {
  x;
  y;
  speed;
  points;

  constructor(x, y, speed) {
    this.x = x;
    this.y = y;
    this.speed = speed;
    this.points = 1;
  }

  show() {
    noStroke();
    fill(“red”);
    rect(this.x, this.y, 20, 20);
  }
}
```

Subclassing in code – betere constructor

```
class RotteAppel extends Appel {
  constructor(x, y, speed) {
    super(x, y, speed)
    this.points = -1;
  }

  show() {
    super.show();
    fill(“brown”);
    rect(this.x, this.y, 20, 20);
  }
}
```

```
class Appel {
  x;
  y;
  speed;
  points;

  constructor(x, y, speed) {
    this.x = x;
    this.y = y;
    this.speed = speed;
    this.points = 1;
  }

  show() {
    noStroke();
    fill(“red”);
    rect(this.x, this.y, 20, 20);
  }
}
```

Subclassing in code – betere constructor

```
class RotteAppel extends Appel {
  constructor(x, y, speed) {
    super(x, y, speed)
    this.points = -1;
  }

  show() {
    super.show();
    fill("brown");
    rect(this.x, this.y, 20, 20);
  }
}
```

Roept de constructor van de superclass aan.

```
class Appel {
  x;
  y;
  speed;
  points;

  constructor(x, y, speed) {
    this.x = x;
    this.y = y;
    this.speed = speed;
    this.points = 1;
  }

  show() {
    noStroke();
    fill("red");
    rect(this.x, this.y, 20, 20);
  }
}
```

Hoe definieer ik een subclass - superclass

```
class <Naam van de class> {
  attribuut1;
  attribuut2;

  constructor(parameter1, parameter2) {
    this.attribuut1 = parameter waarvan je de waarde wilt gebruiken;
    this.attribuut2 = parameter waarvan je de waarde wilt gebruiken;
  }

  methodenaam() {
    // code die uitgevoerd moet worden

    return <waarde>; // alleen als er een waarde teruggegeven moet worden
  }
}
```

Hoe definieer ik een subclass - subclass

```
class <Naam van de class> extends <Naam van superclass> {
  attribuut3; // specifieke attributen voor deze subclass
  attribuut4;

  // constructor heeft de parameters van de superconstructor, PLUS
  // de parameters die nodig zijn voor de constructie van de subclass
  constructor(parameter1, parameter2, parameter3, parameter4) {
    super(parameter1, parameter2);
    this.attribuut3 = parameter waarvan je de waarde wilt gebruiken;
    this.attribuut4 = parameter waarvan je de waarde wilt gebruiken;
  }
  // methodes mogen een methode uit superclass overschrijven of helemaal nieuw zijn.
  methodenaam() {
    // code die uitgevoerd moet worden, gebruik evt. super.methodenaam() als je overschrijft.
    return <waarde>; // alleen als er een waarde teruggegeven moet worden
  }
}
```

Overerving vervolg

Dierentuin

```
class Aap {
  naam;
  leeftijd;

  constructor(naam, leeftijd) {
    this.naam = naam;
    this.leeftijd = leeftijd;
  }

  maakGeluid() {
    console.log(“HoeHoeHaHa”);
  }

  klimInBoom() {
    console.log(naam + “zit in de boom”);
  }
}
```

Dierentuin

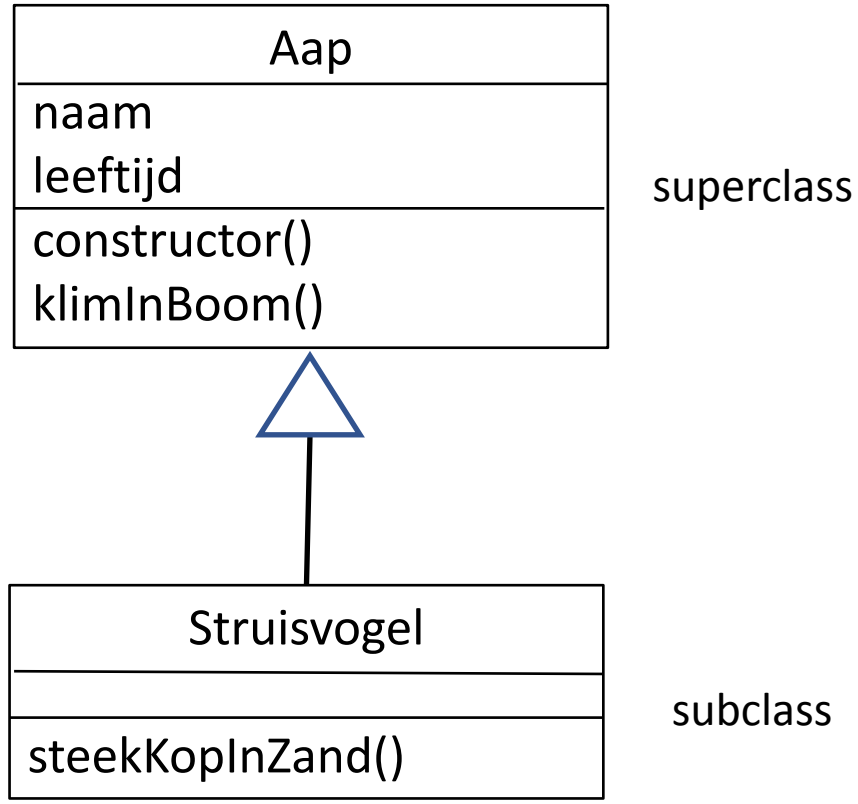
Hoe kun je hier een class hiërarchie van maken?

```
class Aap {
  naam;
  leeftijd;

  constructor(naam, leeftijd) {
    this.naam = naam;
    this.leeftijd = leeftijd;
  }

  maakGeluid() {
    console.log(“HoeHoeHaHa”);
  }

  klimInBoom() {
    console.log(naam + “ zit in de boom”);
  }
}
```

```
class Struisvogel {
  naam;
  leeftijd;

  constructor(naam, leeftijd) {
    this.naam = naam;
    this.leeftijd = leeftijd;
  }

  maakGeluid() {
    console.log(“Groaaaar”);
  }

  steekKopInZand() {
    console(naam + “heeft de kop in het zand gestoken.);
}
```

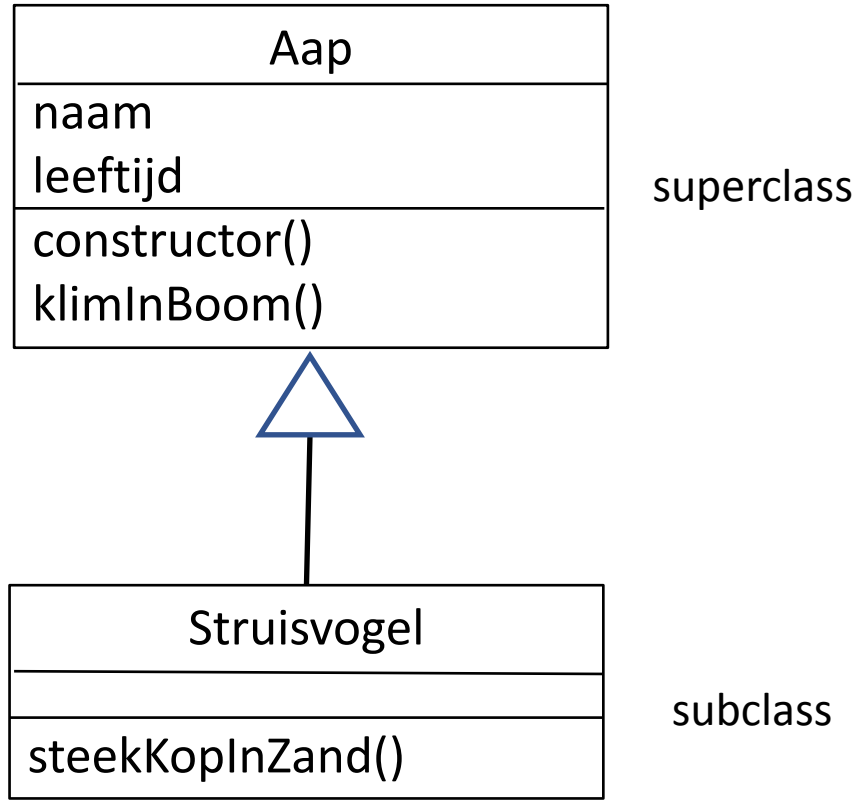
Dierentuin

?


Aap
naam
leeftijd
constructor()
klimInBoom()
superclass
Struisvogel
steekKopInZand()
subclass

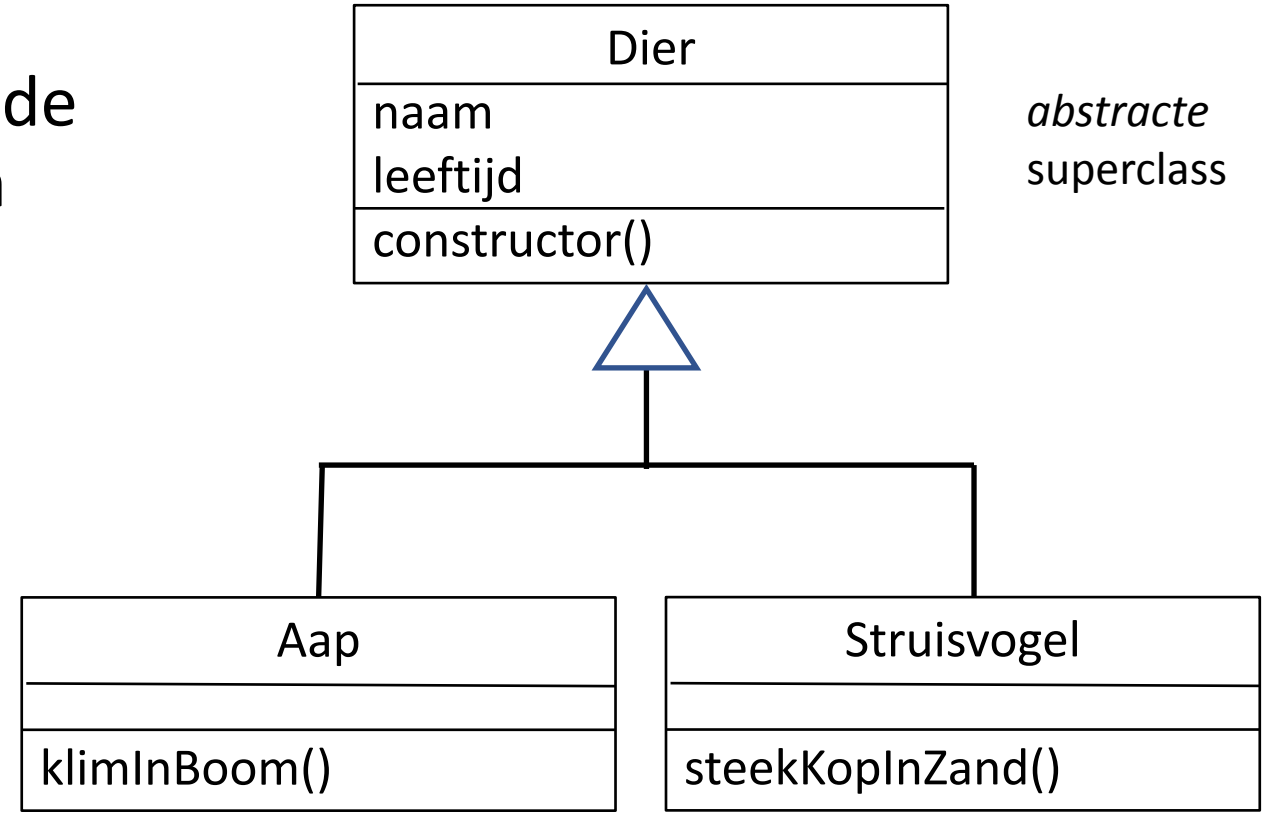
Dierentuin

- Een struisvogel kan niet in een boom klimmen. Volgens het diagram wel
- Een struisvogel is ook helemaal geen speciaal soort aap.



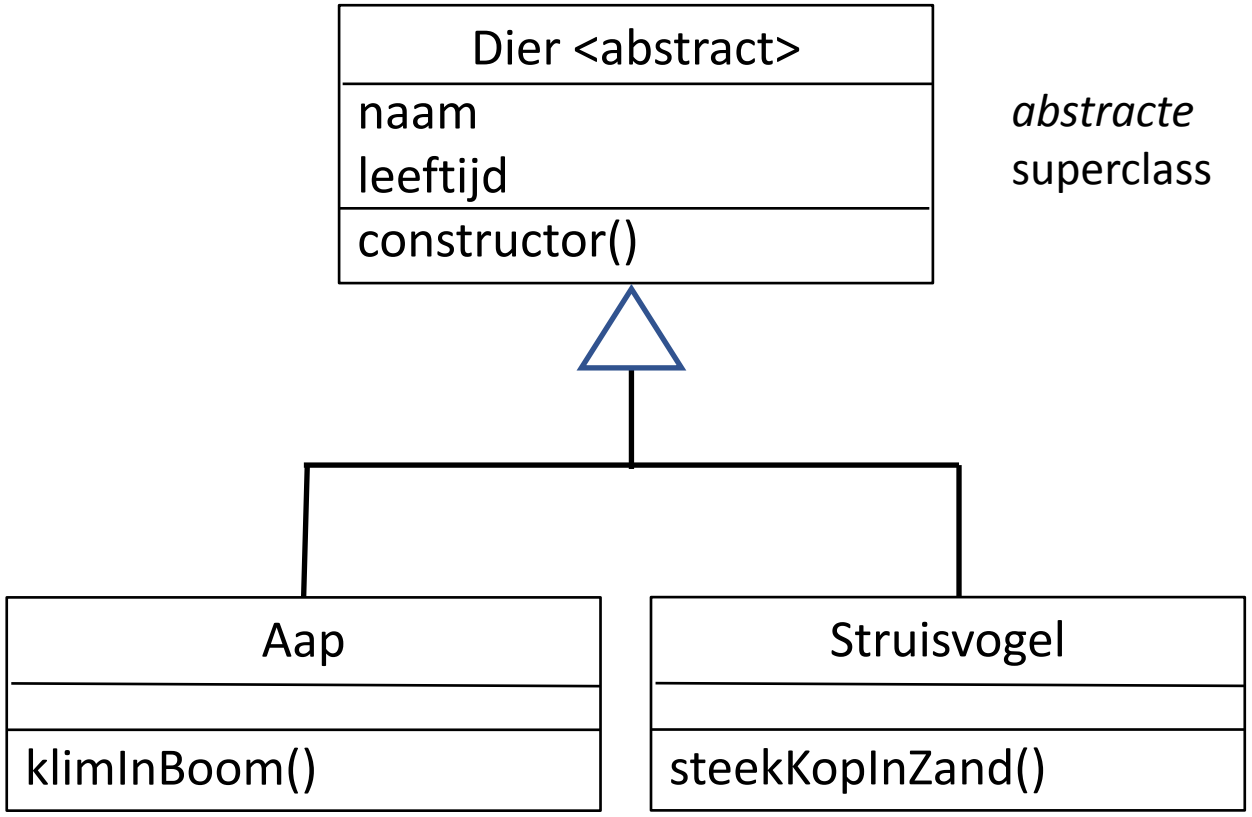
Dierentuin

- Het zijn wel allebei **dieren** uit de dierentuin en hebben daarom gedeelde eigenschappen.



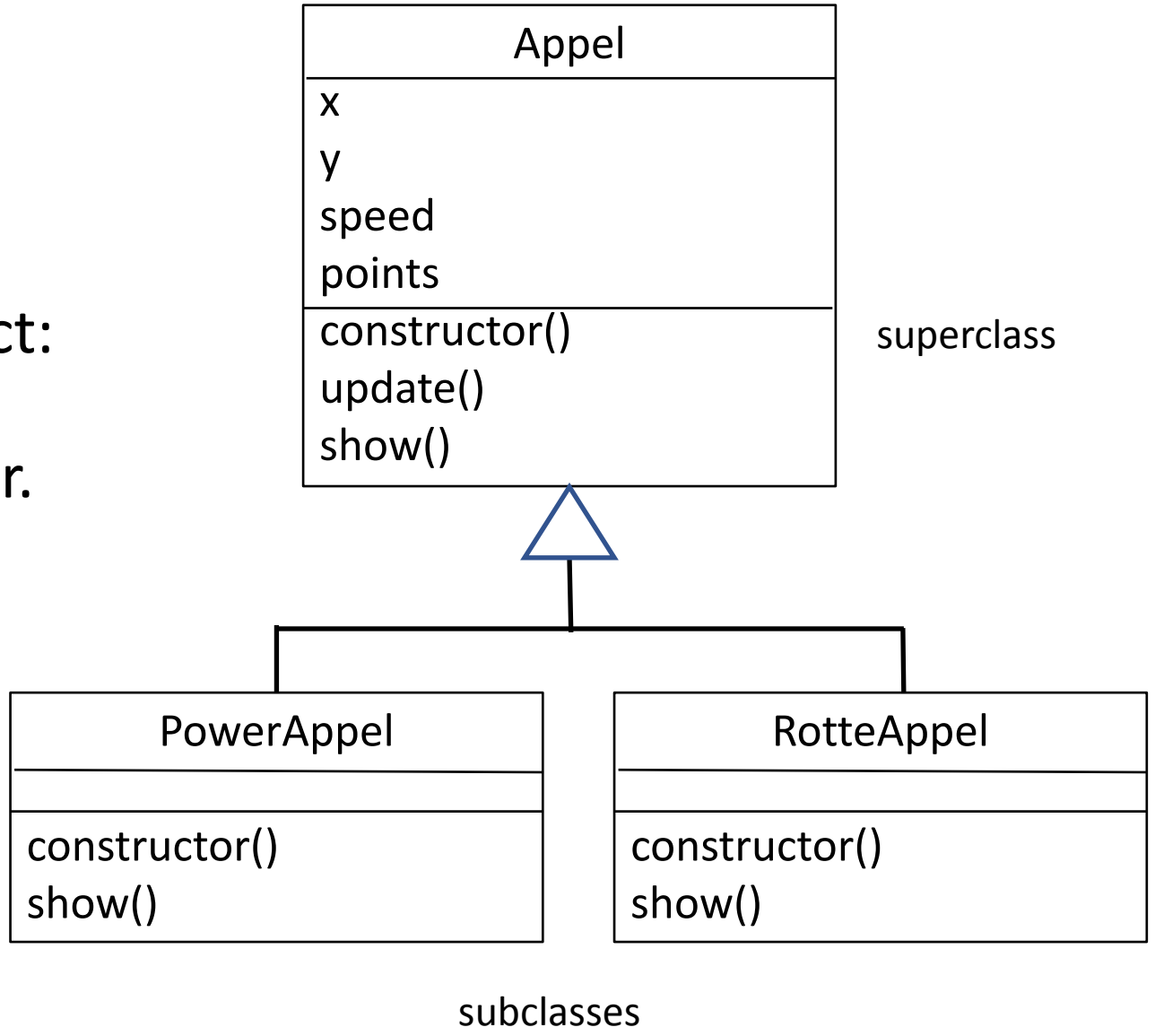
Dierentuin

- Het zijn wel allebei **dieren** uit de dierentuin en hebben daarom gedeelde eigenschappen.
- In onze dierentuin hebben we echter alleen objecten van concrete dieren (Aap , Struisvogel, etc.), een objecten die alleen ‘Dier’ zijn.
- Een class die alleen bedoeld is om gedeelde eigenschappen in op te nemen maar niet bedoeld is om objecten van te maken, heet een **abstracte klasse**.



LET OP

- Een superclass met meerdere subclasses is **niet altijd** abstract: er komen nog steeds appel-objecten in het spel voor.



Inkapseling

Inkapseling

- Soms wil je dat attributen alleen toegankelijk zijn vanuit de **eigen** klasse. Andere klassen mogen dan bijvoorbeeld niet zomaar zo'n attribuut vanuit hun eigen code aanpassen.
- Dit principe heet inkapseling
- Een attribuut dat vrij toegankelijk is voor andere klassen is *public*
- Een attribuut dat niet vrij toegankelijk is voor andere klassen is *private*
- Om private attributen toch toegankelijk te maken kun je daarvoor methodes gebruiken: een setter-methode om een attribuut aan te passen en een getter-methode om een attribuut uit te lezen.
- Waarom handig? -> je kunt zo voorkomen dat iemand jouw klasse gebruikt op een manier die eigenlijk helemaal niet zou moeten kunnen.

Voorbeeldsituaties

- Private attribuut zonder getter of setter-methode: alleen voor gebruik binnen de eigen klasse zelf.
- Private attribuut met alleen een getter-methode -> read only attribuut
- Private attribuut met zowel een getter- als een setter-methode -> je houdt meer controle over het veranderen van een attribuut. Je kun extra aanpassingen maken aan anderen attributen of controleren of jij vindt dat de waarde dit het attribuut moet krijgen wel is toegestaan.

Inkapseling van attribuut ‘punten’

```
class Appel {
  x;
  y;
  speed;
  #points;

  constructor(x, y, speed) {
    this.x = x;
    this.y = y;
    this.speed = speed;
    this.#points = 1;
  }

  getPoints() {              // getter voor points
    return this.#points;
  }
  // hier zouden nog meer methodes moeten staan,
  // zoals show() en update()
}
```

```
class Appel {
  x;
  y;
  speed;
  points;

  constructor(x, y, speed) {
    this.x = x;
    this.y = y;
    this.speed = speed;
    this.points = 1;
  }

  show() {
    noStroke();
    fill(“red”);
    rect(this.x, this.y, 20, 20);
  }
}
```

Inkapseling van attribuut ‘punten’

```
class Appel {
  x;
  y;
  speed;
  #points;

  constructor(x, y, speed) {
    this.x = x;
    this.y = y;
    this.speed = speed;
    this.#points = 1;
  }

  getPoints() {              // getter voor points
    return this.#points;
  }
  // hier zouden nog meer methodes moeten staan,
  // zoals show() en update()
}
```

```
var appeltje = new Appel(20, 50, 3);

// je kunt nu niet meer direct het
// attribuut points van appeltje
// opvragen, maar via gettermethode:
console.log(appeltje.points);
console.log(appeltje.getPoints());

// veranderen is niet mogelijk:
appeltje.points = 5;
```

Inkapseling om setter te laten beschermen

```
class Appel {
 x;
 y;
 #speed;
 #points;

 constructor(x, y, speed) {
  this.x = x;
  this.y = y;
  setSpeed(speed);
  this.#points = 1;
 }

 getSpeed() {              // getter voor speed
  return this.#speed;
 }

 setSpeed(speed) {
  if (speed >= -10 && speed <= 10) {
   this.#speed = speed;
  }
  else {
   // wat zou je hier doen?
  }
 }
 // hier zouden nog meer methodes moeten staan,
 // zoals getPoints(), show() en update()
}
```

Stel je wilt dat de speed nooit minder dan -10 of hoger dan 10 wordt…

Inkapseling om setter te laten beschermen

```
class Appel {
  x;
  y;
  #speed;
  #points;

  constructor(x, y, speed) {
    this.x = x;
    this.y = y;
    setSpeed(speed);
    this.points = 1;
  }

  getSpeed() {              // getter voor speed
    return this.speed;
  }

  setSpeed(speed) {         // setter voor speed
    if (speed >= -10 && speed <= 10) {
      this.speed = speed;
    }
    else {
      // wat zou je hier doen?
    }
  }
  // hier zouden nog meer methodes moeten staan,
  // zoals getPoints(), show() en update()
}
```

```
getSpeed() {       // getter voor speed
  return this.#speed;
}

setSpeed(speed) {  // setter voor speed
  if (speed >= -10 && speed <= 10) {
    this.#speed = speed;
  }
  else {
    // wat zou je hier doen?
  }
}
```

Hoe toegang schematisch weer?

algemeen

KlasseNaam
+ditIsEenPublicAttribute : string -ditIsEenPrivateAttribute : string geboortedatum : date
geeftLeeftijd() : number setterVanPrivateAttribute(s : string) getterVanPrivateAttribute() : string

voorbeeld

Leerling
-leerlingnummer : number -voornaam : string -achternaam : string +notitie : string -geboortedatum : date
geefLeeftijd() : number getGeboortedatum() : date getLeerlingnummer() : number getVoornaam() : string setVoornaam(s : string) getAchternaam() : string setAchternaam(s : string)

Controleer voor jezelf:

- leerlingnummer en geboortedatum zijn readOnly
- notitie is als enige public
- voornaam en achternaam zijn private, maar kunnen met een setter wel worden veranderd. Dit heeft als voordeel dat de setter eerst nog kan controleren of er bijv. geen lege string ("") wordt meegegeven

Inkapseling i.c.m. subclasses

```
class Appel {
  x;
  y;
  speed;
  #points;

  constructor(x, y, speed) {
    this.x = x;
    this.y = y;
    this.speed = speed;
    this.#points = 1;
  }

  getPoints() {              // getter voor points
    return this.#points;
  }
  // hier zouden nog meer methodes moeten staan,
  // zoals show() en update()
}
```

```
class SuperAppel extends Appel {

  constructor(x, y, speed) {
    this.x = x;
    this.y = y;
    this.speed = speed;
    this.#points = 5;
  }

  show() {
    noStroke();
    fill("purple");
    rect(this.x, this.y, 20, 20);
  }
}
```

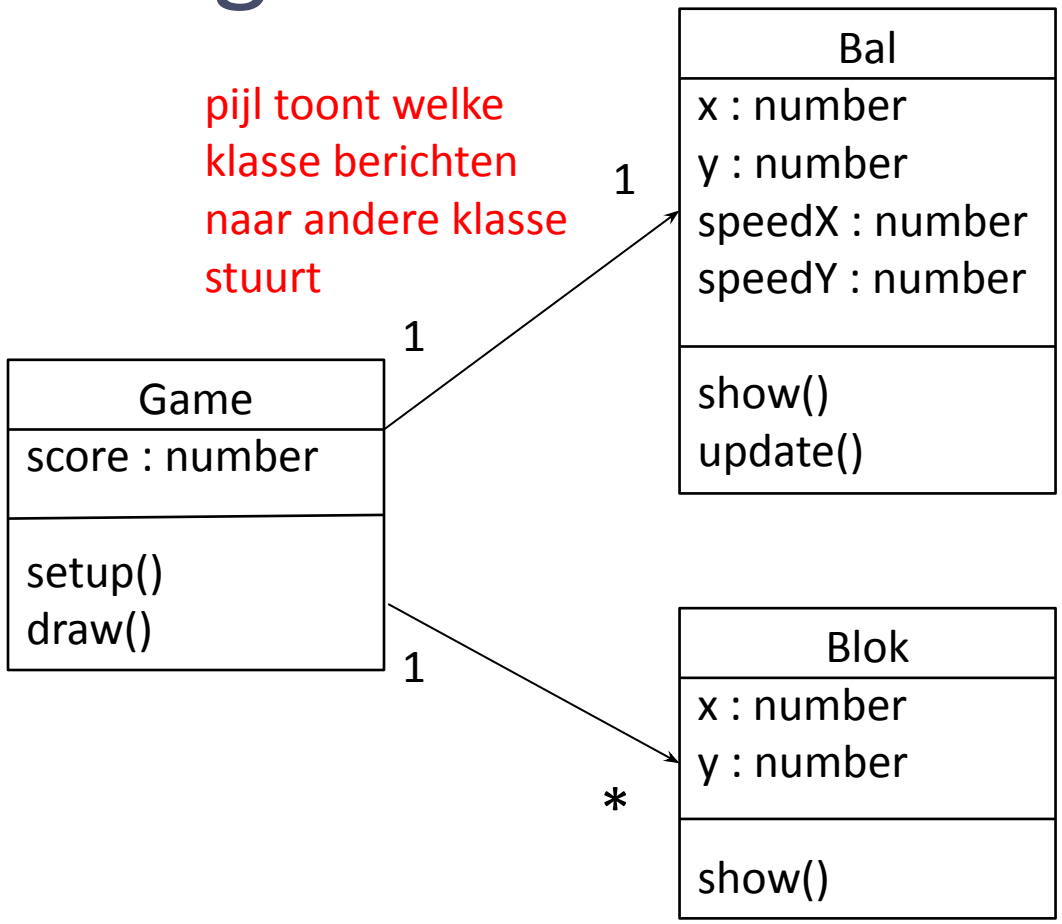
Een SuperAppel heeft meer punten, maar waarom kan dat niet op deze manier?

Associatie

Associatie

- Tot nog toe bevatte een class m.b.v. attributen data zoals string, number, boolean, of een array (=mogelijk meerdere stukjes data waarnaar een attribuut verwijst
- Je kunt in een class ook gebruik maken van objecten van een andere class.
- Hiermee kun je bijv. complexe functionaliteit netjes opsplitsen in afgebakende klassen.

Associatie in diagram



Associatie in code

```
class Game {
  score;
  bal;
  blokken;

  constructor() {
    this.score = 0;
    this.blokken = [];
    this.blokken.push() = new Blok();
    this.bal = new Bal(50, 200);
  }
}
```