Dagens föreläsning

Klasser
och
arrayer

Medan ni väntar: *Skriv en metod som tar emot 4 heltalsparametrar och returnerar värdet av den minsta av dessa!*

Introduktion till **klasser**

Ett Javaprogram består av en eller flera *klasser*.

Klassen är den minsta självständiga enheten i ett Javaprogram.

Vilka klasser har ni sett/använt?

Klasser har *namn*. Hur ser man om ett namn står för en klass?

En klass kan representera

- En konkret *fysisk* enhet. Exempel: fordon, tärning, person, hus, kondensator, sköldpadda, biljardboll, ...
- En mer *abstrakt* enhet. Exempel: triangel, bankkonto ...
- En *programmeringsteknisk* enhet. Exempel: program, teckenström, `Scanner`, `Math` ...

Vad är en klass?

- En *textfil* med typen `.java`.
- En *programmodul*.
- En abstrakt *beskrivning* av någon typ objekt.
- En *mall*.
- En *datatyp*.

Klasser och objekt

En typisk klass konstrueras med

- *instansvariabler*
- *konstruktorer*
- *metoder*

Instansvariablerna står för någon *egenskap* (ex: *längd*, *vikt*, *färg*, *namn*, . . .)

Metoderna står för någon form av operation (`move`, `turn`, . . .) eller används för att hämta information (`getXPos`, `getWorld`, . . .)

Fram t o m nätlektion 4 har ni sett metoder men inga instansvariabler.

De variabler ni sett har varit *lokala* variabler eller *formella parametrar*.

Klassen Turtle

```
public class Turtle {
  public static final
           double RADIUS = 10.0;
  private int x;
  private int y;
  private int direction;
  private float size = 1.0f;
  private Color color;
  private Color limbColor;
  private boolean visible;
  private boolean drawPathFlag;
  private World world;

  public Turtle(World w){...}
  public Turtle(World w,
                int x,
                int y){...}

  public World getWorld(){...}
  public int getXPos(){...}
  public int getYPos(){...}
  public int getDirection(){...}
  public void setDirection(int dir){...}  }
```

```
public double getSize(){...}
public void setSize(double size){...}
public double getRadius(){...}
public void setRadius(double radius){..
public Color getColor(){...}
public Color getLimbColor(){...}
public void setColor(int red,
                     int green,
                     int blue) {...}
public boolean isVisible(){...}
public void setVisible(boolean visible)
public boolean isPathEnabled(){...}
public void enablePath(){...}
public void disablePath(){...}
public double distanceTo(int x, int y){
public String toString(){...}
public void moveTo(int x, int y){...}
public void turn(int degrees){...}
public void turnTo(int x, int y){...}
public void turnTo(Turtle t){...}
public void move(int step){...}
```

Klasser

Klasser *instansieras* till *objekt*. Detta görs med operatorn **new**.

Referensvariabler (pekare, adresser) används för att hålla reda på objekt.

Operatorn **new** returnerar en referens till det skapade objektet.

För att komma åt attribut och metoder i används *punktnotation*.

Instansvariabler och metoder har en *synlighet* som anges med **public** eller `private` (två till finns).

Metoderna är ofta publika och instansvariablerna privata.

Exempel: En klass för att representera en tärning

Vilka egenskaper har en tärning?

- *Färg*?
- *Vikt*?
- *Material*?
- *Storlek*?
- *Antal sidor*?
- *Aktuellt värde*?
- ...

Beror på vad vi vill göra!

Nöjer oss med *Antal sidor* och *Aktuellt värde*

Vilka operationer vill vi ha?

- *Skapa* en tärning
- *Slå* en tärning
- *Avläsa* tärningens värde
- *Ändra* tärningens värde? Nej, knappast.
- *Ändra* antalet sidor? Möjligen.
- ...

En första skiss av klassen `Dice`

```
public class Dice {
   private int nSides;
   private int value;

   public Dice() { ... }

   public int getValue() { ... }

   public void roll() { ... }
}
```

Två *instansvariabler*

En *konstruktor*

Två *metoder*

Konstruktorer

Har samma namn som klassen. I detta fall `Dice`.

Används när ett objekt ska skapas.

Ger instansvariabler värden:

```
public Dice() {
    this.nSides = 6;
}
```

Fråga: Hur gör man för att få en tärning med ett annat antal sidor?

Alternativ konstruktor

För att skapa en tärning med ett annat antal sidor än 6:

```
public Dice(int nSides) {
    this.nSides = nSides;
}
```

Obs: Här är det nödvändigt att använda `this`!

Nu kan man skapa tärningar:

```
Dice d1 = new Dice();
Dice d2 = new Dice(17);
Dice d3 = new Dice(40396);
```

Måste man skriva konstruktorer?

Dice: Metoderna

Definition av metoderna:

```
public int getValue() {
   return value;
}


public void roll() {
   value = (int)(Math.random()*nSides) + 1;
}
```

Förbättringar av konstruktorerna

Vad får `value` för värde när en tärning skapas?

Defaultvärde är på instansvariabler är 0 (eller `null` för referensvariabler).

Fråga: Vilket värde borde `value` få?

Så här då?

```
public Dice() {
    this.nSides = 6;
    this.roll();
}

public Dice(int nSides) {
   this.nSides = nSides;
   this.roll()
}
```

Fler förbättringar av konstruktorerna

Vad händer om man gör `new Dice(-3)`?

Konstruktorerna bör kontrollera att den får vettiga värden!

```
public Dice(int nSides) {
    if (nSides < 2) {
        System.out.println("Dice: Wrong number of sides");
        this.nSides = 6;
    }
    this.nSides = nSides;
    this.roll()
}
```

Dice: Ännu fler förbättringar

Upprepning av identisk kod är en styggelse!

Den parameterlösa konstruktorn skrivs snyggare så här:

```
public Dice() {
    this(6);
}
```

Uttrycket `this( ... )` i en konstruktor anropar alltså en annan konstruktor.

Flexiblare användning med modifierad `roll`-metod:

```
public int roll() {
    this.value = (int)(Math.random()*nSides) + 1;
    return this.value;
}
```

Dice: toString-metod

Metoden `toString` definierar en konvertering från ett objekt till ett `String`-objekt. Den vill man så gott som alltid ha!

```
public String toString()  {
   return  "" + value;
}
```

Öh?

eller

```
public String toString()  {
    return "Dice(" + nSides + ", " + value + ")";
}
```

Råd: Gör inte `toString`-metoderna så pratiga!

Arrayer

En *array* är en datatyp som representerar *flera* värden av samma typ.

Exempel:

värden	7	23	8	1	4	17	9	3	42
index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*

I Java är arrayer *objekt* som skapas med **new**.

Uttrycket `new int[9]`

skapar följande array-objekt

värden	0	0	0	0	0	0	0	0	0
index	*0*	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*

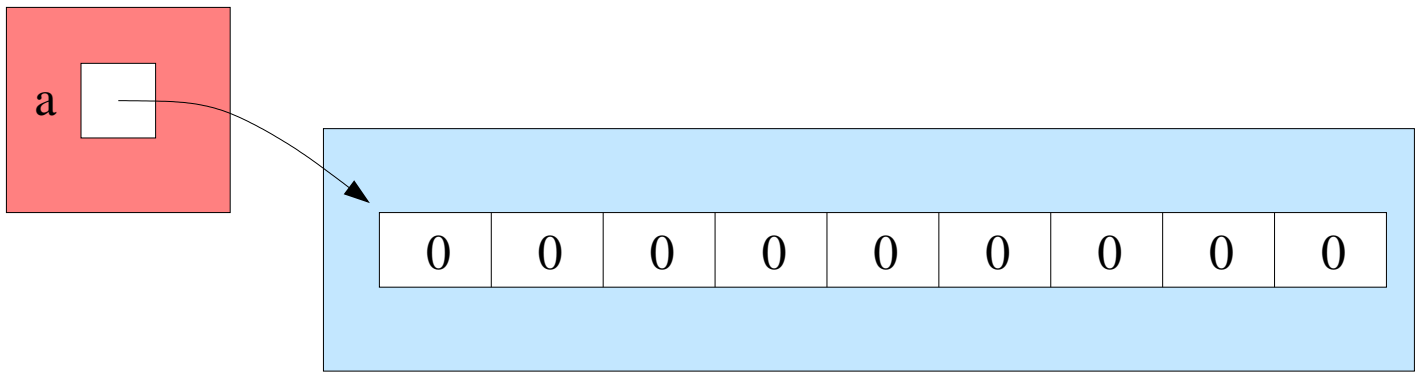
Arrayer

För att hålla reda på array-objekt användes referensvariabler.

Satsen

```
int[] a = new int[9];
```

skapar följande bild:

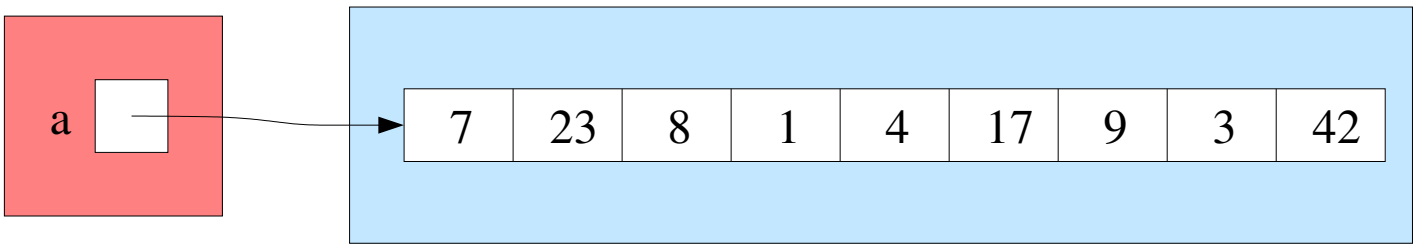


Arrayer

Det går att ge värden till elementen i samband med att det skapas. Satsen

```
int[] a = new int[]{7, 23, 8, 1, 4, 17, 9, 3, 42}
```

skapar det array-objekt och sätter `a` att referera det.



Skrivs dock vanligen (läs alltid) utan `new int[]` dvs så här

```
int[] a = {7, 23, 8, 1, 4, 17, 9, 3, 42}
```

Det är bara vid *deklarationer* av array-referenser man kan göra på detta sätt.

Arrayer

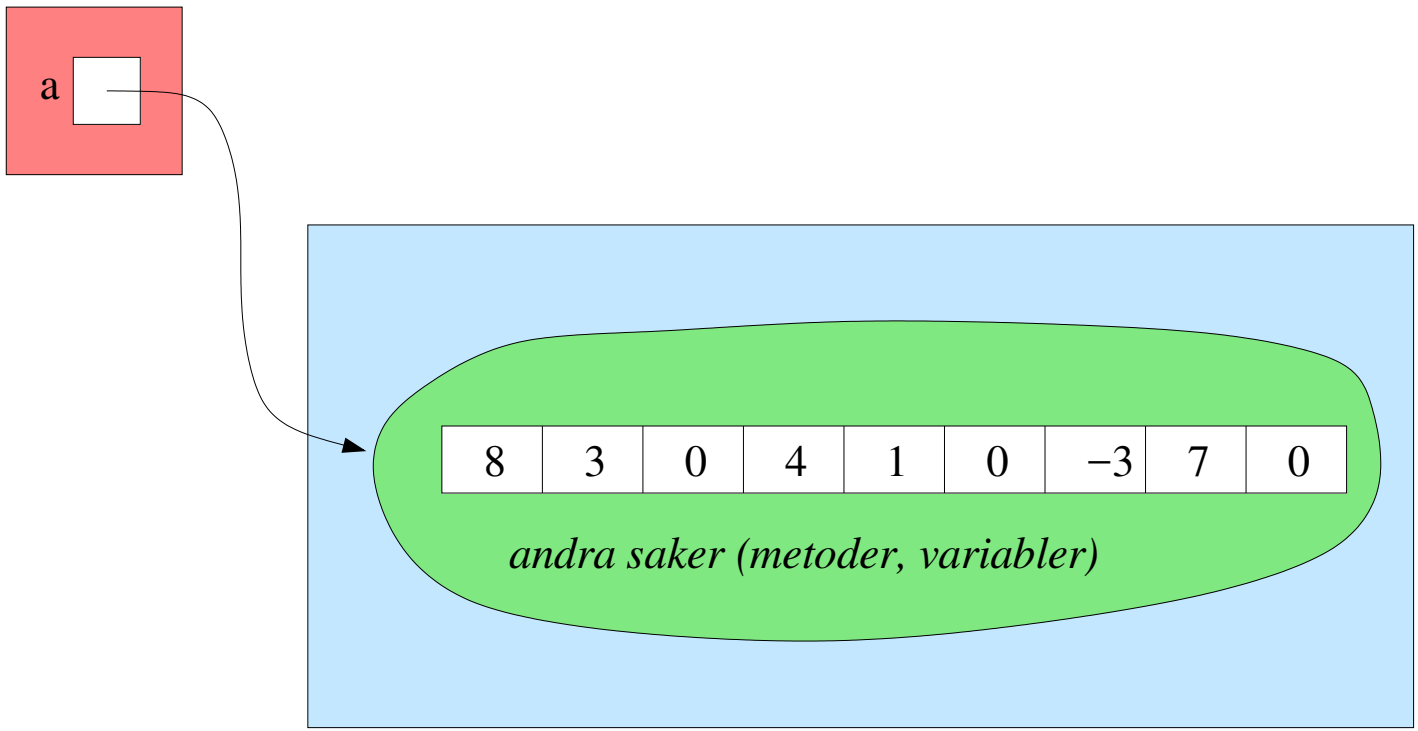
För att komma åt enskilda element i arrayen används *index* inom `[ ]`-parenteser.

Exempel:

```
a[0] = -47;
a[1] = 3;
a[2] = 5;
a[3] = a[1] + a[2];
a[a[1]] = (int)(Math.random()*1000);
a[(int)(Math.random()*9)] = 99;
char c = 'd';
a[c-'a']++;
```

Arrayer

Array-objekt innehåller fler saker än värden. En mer rättvisande bild är



Arrayer

Ett exempel på information som finns i array-objekten är attributet `length` (obs: *ej* en metod)

Exempel: Räkna antalet negativa tal i arrayen `a`:

```
int n = 0;
int i = 0;
while (i < a.length) {
    if (a[i] < 0) {
       n++;
    }
    i++;
}
System.out.println("Number of negative values: " + n);
```

Sammanfattning av arrayers egenskaper

- En array innehåller 0 eller flera element.
- Alla element måste vara av samma typ.
- Elementen kan vara av vilken typ som helst inklusive objektreferenser och arrayer.
- Ett skapat array-objekt har en fix storlek som inte kan ändras men en array-referens kan tilldelas ett annat array-objekt med annan storlek.
- Attributet `length` innehåller arrayens storlek.
- Man använder index-operatorn `[ ]` för att referera enskilda element.
- Indexering med `[ ]` är en mycket effektiv operation.

for-satsen

För att iterera över elementen i en array används ofta **for**- i stället för **while**-satsen.

Exempel: Räkna antalet negativa värden i en array

```
int n = 0;
for (int i=0; i<a.length; i++) {
   if (a[i] < 0) {
      n++;
   }
}
```

```
int n = 0;
int i = 0;
while (i<a.length) {
   if (a[i] < 0) {
      n++;
   }
   i++;
}
```

Funktionen är identisk (men loopvariabeln `i` existerar inte efter **for**-loopen)

”for-each”-satsen

Exempel: Räkna antalet negativa element som ovan

```
int n = 0;
for (int i = 0; i < a.length; i++) {
   if (a[i] < 0) {
        n++;
   }
}
```

```
int n = 0;
for (int v : a) {
   if (v < 0) {
      n++;
   }
}
```

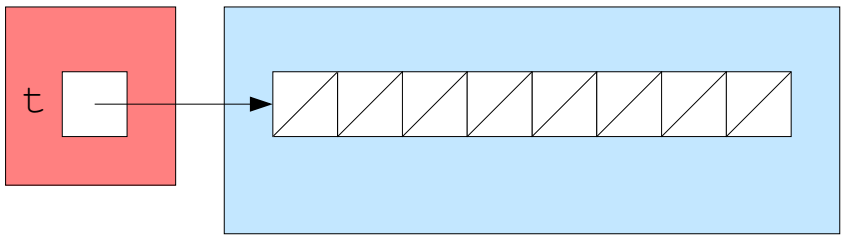
Observera att

- Variabeln `v` kommer successivt att anta *alla värden* som finns i arrayen
- Kommer inte åt några *index*

Arrayer kan lagra objektreferenser

Exempel: Array med paddor.

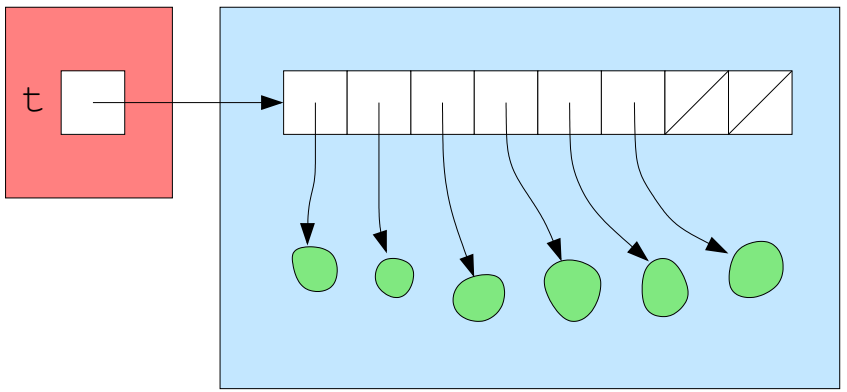
```
Turtle[] t = new Turtle[8];
```



Skapar en array men inga paddor. Ett snedstreck anger `null`-referens.

Vi kan skapa paddor på vanligt sätt och lägga in dem i världen:

```
for (int i = 0; i < 6; i++) {
    t[i] = new Turtle(w);
}
```



(Förutsätter att `w` refererar en `World`.)

Programmeringsexempel

Ett program med flera paddor som irrar omkring och krockar.

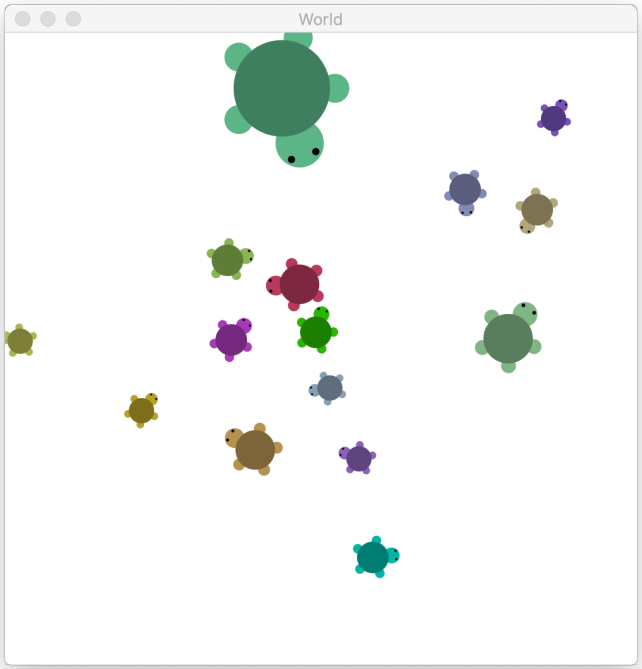
Tanken är att något ska hända när de krockar.

Har valt ”kanibalpaddor” dvs de äter upp varandra när de krockar.

Min vision:

Större paddor äter upp mindre och uppätna paddor försvinner.

Efter ett tag ska det se ut så här:



”Startapproximation”

```
public static void main(String[] args)
       throws InterruptedException  {
   World w = new World(500,500);
   Turtle[] t = new Turtle[50];

   for (int i = 0; i < 50; i++) {
      t[i] = new Turtle(w);
   }

   while ( true ) {
      Thread.sleep(50);
      for (int i = 0; i<50; i++) {
          t[i].moveRandom();
      }
   }
}
```

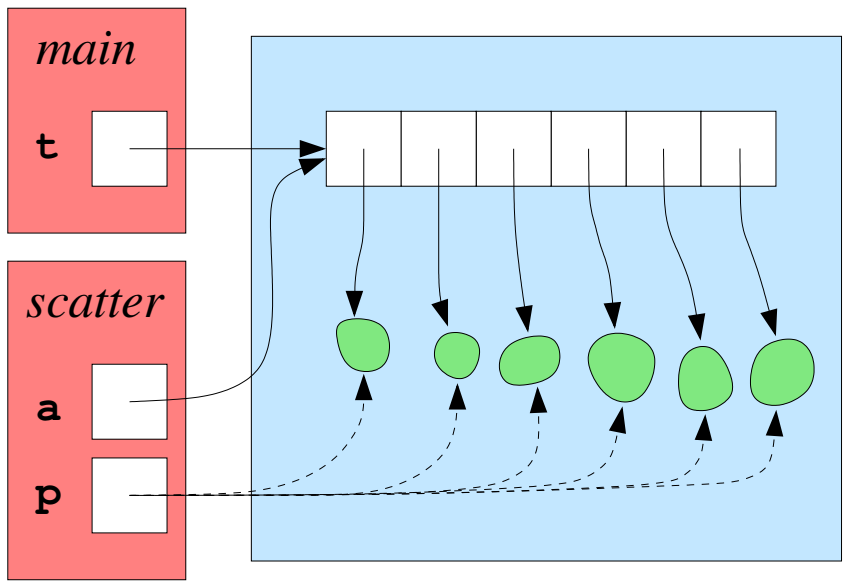
Önskade ändringar

▶ Lättare att ändra antal, världsstorlek och sovtidstorlek!	Använd variabler i stället för konstanter!
▶ För mycket ”random”!	Ny `Turtle`-metod: `dizzyMove`
▶ ”Studs” mot världens kanter!	Kan göras i `dizzyMove` m.h.a. en `distanceBorder`
▶ Ta bort spåren!	Enklast när paddorna skapas
▶ Utplaceringen slumpvis över världen!	En `scatter`-metod (Så får vi också ett exempel på array-parameter)

Arrayer som parametrar

Antag att vi skapar en array med 6 paddor i `main` och anropar metoden `scatter` med arrayen `t` som parameter.

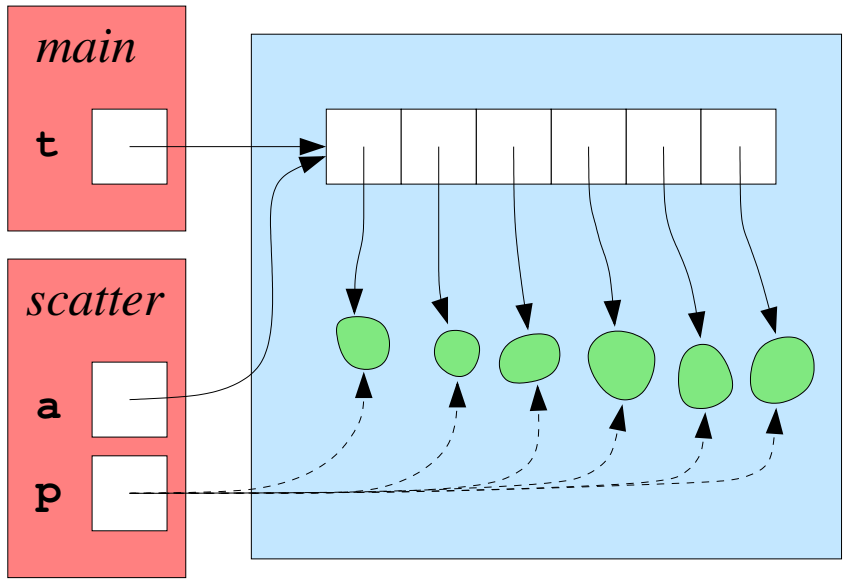
```
public static void
     scatter(Turtle[] a) {
  World w = a[0].getWorld();
  int s = w.getWidth();
  for (Turtle p : a) {
    p.moveTo((int)(Math.random()*s),
             (int)(Math.random()*s));
    p.turn((int)(Math.random()*360));
  }
}
```



Arrayer som parametrar

Observera att metoden `scatter`

- kan ändra i `Turtle`-objekten,
- skulle kunna ändra i arrayen (t ex ta bort eller lägga till `Turtle`-objekt) men
- kan inte ändra på `t` eller någon annan variabel i `main`-metoden.



Fortsättning på programmeringsexemplet

▶ Kontrollera om paddor krockar	En *dubbelloop* och en metod `collide`
▶ Ta bort en av de kolliderande paddorna	Metoden `remove` i klassen `World` samt nulla i arrayen. Se upp med `NullpointerException`!
▶ Se till att den större paddan växer	Använd `getSize` och `setSize`
▶ Avbryt när bara en padda kvar.	En `countTurtles` som räknar paddor i arrayen

Länkar till koden med `Turtle` som innehåller några nya metoder samt till `TurtleMess`.