Rekursiva funktioner

Algoritmer och datastrukturer

Obligatorisk Laboration nr 5

Syfte

Att ge träning i problemlösning med rekursion.

Mål

Den som självständigt av egen kraft löst uppgifterna i denna laboration bör ha uppnått en god förmåga att konstruera rekursiva algoritmer.

Litteratur

Weiss kap. 7, Wikipedia.
Nyckelord: Fraktal, Koch snowflake, Mandelbrot, Polygon.

Programkod

Givna kodavsnitt finns på kursens hemsida.

Genomförande och redovisning

Uppgifterna är obligatoriska.

- Redovisning sker i grupper om två personer – inte färre, inte fler.
- Varje grupp skall självständigt utarbeta sin egen lösning.
  Samarbete mellan grupperna om övergripande principer är tillåtet, men inte plagiering.

Gå till kurshemsidan och vidare till **Laborationer->Hur man använder Fire-systemet**.

- Om du inte registrerat labbgrupp i Fire än: Gör laboration 0.0 innan du fortsätter!
- Redovisa filerna `LinearRecursion.java`, `Mobile.java`, `ConcaveKoch.java`, `Penta.java`, `Square.java`, samt `README.txt` med allmänna kommentarer om lösningen.
- Andra halvan av **Redovisningsproceduren** beskriver hur du postar materialet i Fire.
- Senaste redovisningsdag, se Fire.

Allmänt

Alla funktioner skall vara *rekursiva*! Lämplig arbetsgång:

1. Bestäm vad funktionen skall göra i basfallet.
2. Rekursionssteget
   a) Formulera ett rekursionsantagande – skriv ner det!
   b) Konstruera lösningen med hjälp av resultatet(n) från de(t) rekursiva anropet(n) med stöd av rekursionsantagandet.

Uppgifterna är inte ordnade i svårighetsgrad. Ett problem som är lätt för dig kan vara svårt för någon annan och vice versa. Kör inte fast på någon uppgift. *Om du inte kommer på lösningen inom rimlig tid så gå hellre vidare och återvänd till problemet senare. Förmågan att tänka rekursivt måste du träna upp själv. Det är som med höjdhopp, du kan kanske hoppa en aning högre genom att titta på en duktig höjdhoppare, men det räcker inte högt.*

A. Linjär rekursion över heltal och listor m.m.

Metodstubbar och testfall finns i filen **`LinearRecursion.java`**.

Uppgift A.1

Skriv en rekursiv funktion som läser en indatarad tecken för tecken och skriver ut raden baklänges. Funktionen skall ha signaturen

```
public static void reverseInput()
```

och den får endast deklarera *en* lokal variabel av typen **`char`** för att lagra tecken. Använd

```
(char)System.in.read()
```

för inläsningen. När sista tecknet i raden lästs returnerar `read` radslutstecknet `'\n'`.
Ex. Indataraden

`rekursion` *return*

skall ge utskriften

`noisruker`

Skulle du mot förmodan få utskriften **`noitareti`** har du gjort fel.

Tips: Rita en liten figur över en indatarad. Hur skall rekursionsantagandet se ut? Formulera det med stöd av figuren. Vilka tecken kan du anta att det rekursiva anropet tar hand om? Vad skall du göra med resten? - *tar det stopp så prova nästa uppgift så länge!*

Uppgift A.2

Konstruera en rekursiv funktion som utför multiplikation av två heltal genom upprepad addition genom att utnyttja de aritmetiska sambanden

$$0 \times n = 0$$
$$(m + 1) \times n = n + (m \times n)$$
$$(-m) \times n = -(m \times n)$$

funktionen skall ha signaturen

```
public static int multiply(int m,int n)
```

och den får endast använda operatorerna + och -, men givetvis inte `*`.

Uppgift A.3

Konstruera en rekursiv funktion som returnerar antalet decimala siffror i ett ickenegativt heltal. Funktionen skall ha signaturen

```
public static int countDigits(int n)
```

och den får endast använda operatorerna + och / (heltalsdivision). Glöm inte att 0 är ett ensiffrigt tal!

Listor

Listorna i de följande exemplen saknar huvudnoder eftersom dessa oftast är överflödiga vid rekursiv listhantering. En tom lista representeras alltså med **`null`**. För enkelhets skull arbetar vi med heltalslistor.

```
public class ListNode {
    public int element;
    public ListNode next;

    public ListNode(int element,ListNode next) {
        this.element = element;
        this.next = next;
    }
}
// Lägg till ett nytt element i början av l [1]
public static ListNode cons(int element,ListNode l) {
    return new ListNode(element,l);
}
```

Exempel: `ListNode l = cons(1,cons(2,cons(3,null)));`

Exempel: Följande rekursiva metod beräknar längden hos en lista:

```
public static int length(ListNode l) {
    if ( l == null )
        return 0;
    else
        return 1 + length(l.next);
}
```

Uppgift A.4

Konstruera en rekursiv funktion som returnerar en kopia av en lista. Funktionen skall ha signaturen

```
public static ListNode copy(ListNode l)
```

och den skall returnera en ny likadan lista med samma innehåll som `l`. *Tips*: Använd `cons`.

Uppgift A.5

Konstruera en rekursiv funktion som givet två listor returnerar en ny lista som innehåller båda listornas element. Funktionen skall ha signaturen

```
public static ListNode append(ListNode l1,ListNode l2)
```

Till exempel skall

```
append(cons(1,cons(2,cons(3,null))), cons(4,cons(5,null)))
```

returnera en lista med likadant innehåll som

```
cons(1,cons(2,cons(3,cons(4,cons(5,null)))))
```

Tips: Definiera funktionen med rekursion över den första listan och utnyttja funktionen `copy` i den förra uppgiften på lämpligt sätt.

[1] En motsvarighet till denna funktion finns i språket LISP. Cons är en förkortning av construct.

B. Förgreningsrekursion över träd (se förel. 10) `Mobile.java`

Klassdefinitionen för `Mobile` finns på nästa sida. Studera, provkör och experimentera med programmet, ändra vikterna lite och kontrollera att balansfunktionen ger önskat resultat.

Uppgift B.1

Lägg till en rekursiv metod i mobilklassen som returnerar mobilens höjd (= antal nivåer). Antag att höjden för en enkel mobil är 1. Testa på representativa fall!

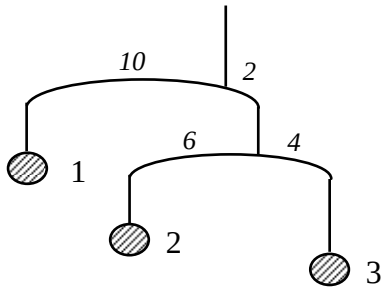
```
public int getHeight()
```

Uppgift B.2 "Avlöva" en mobil

Lägg till en rekursiv metod som skriver ut mobilens "löv". T.ex. skall mobilen `m` skrivas ut som `1 2 3`. Utskriften skall alltså bortse från mobilens struktur och skriva ut den "tillplattad". Vad behöver ändras för att istället få löven utskrivna i omvänd ordning?

```
public void flatten()
```

`m`



Gör sedan en funktion lik föregående men som dessutom visar strukturen med parenteser så att M skrivs ut som `[(1),10,[(2),6,(3),4],2]`. Utskriftsformatet för en enkel mobil är alltså '**(**' *vikt* '**)**', och för en sammansatt '**[**' *mobil* '**,**' *längd* '**,**' *mobil* '**,**' *längd* '**]**'.

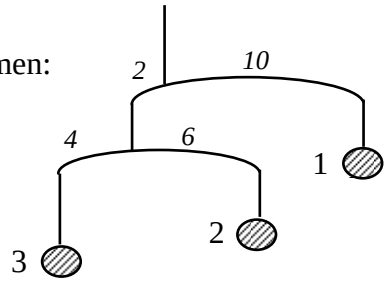
```
public void prettyPrint()
```

Uppgift B.3 Spegla en mobil

Lägg till en rekursiv metod som gör om ett mobilobjekt till sin spegelbild.

```
public void mirror()
```

Efter anropet `m.mirror()` skall `m` ha formen:



Likhet och kloning i Java

Leta upp klassen `Object` i Java API och studera metoderna `equals` och `clone`, speciellt vilka krav de skall uppfylla när de överskuggas i subklasser. Observera att parametertypen (högeroperanden) hos `equals` alltid skall vara `Object` vid överskuggning. Exempel på `equals`-överskuggningar finns i OH från F2.

Uppgift B.4 När är två mobiler lika?

Lägg till metoden `equals` i mobilklassen. Tänk först igenom vad likhet mellan mobiler bör innebära. Tips: Undvik liksom i `isBalanced` att jämföra flyttal med `==`.

```
public boolean equals(Object rhs)
```

Uppgift B.5 Kloning av mobiler

Lägg till en rekursiv metod som returnerar en unik kopia av ett mobilobjekt:

```
public Mobile clone()
```

Om man definierar

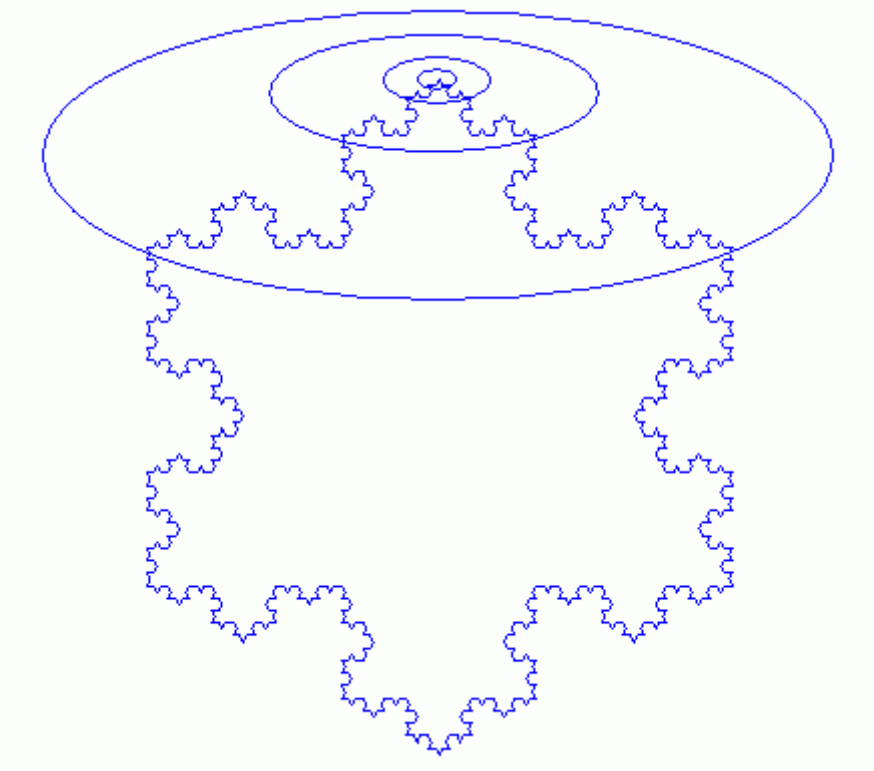
```
Mobile m = new Mobile(...);
```

så skall `m.equals(m.clone())` vara sant, men `m` och klonen vara två unika objekt, d.v.s. `m != m.clone()`. Kontrollera att dina metoder uppfyller detta!

```
public class Mobile {
      private enum MobileType {SIMPLE,COMPOSITE}
      private MobileType type;
      private float weight;                   // Simple
      private float leftLength, rightLength;  // Composite
      private Mobile left, right;

      // Simple case
      public Mobile(float weight) { ... }

      // Composite case
      public Mobile( Mobile left, float leftLength,
                     Mobile right, float rightLength ) { ... }

      // Return the total mass of the mobile
      public float getWeight() { ... }

      // Return the maximum height of the mobile
      public int getHeight() { ... }

      // Print the leaves of the mobile
      public void flatten() { ... }

      // Print a structured view of the mobile
      public void prettyPrint() { ... }

      // Determine if this mobile is balanced
      public boolean isBalanced() { ... }

      // Change this mobile to its mirror image
      public void mirror() { ... }

      public boolean equals(Object rhs) { ... }

      // Return a clone of this mobile
      public Mobile clone() { ... }

      // Determine if this mobile is simple
      private boolean isSimple() { ... }
}
```

C. Fraktaler

En fraktal kurva har egenskapen att förstorade detaljer uppvisar samma struktur som helheten. Exempel på sådana objekt är snöflingekurvor, t.ex. Koch-flingor (efter upphovsmannen Helge von Koch). Kochs kurva ser ut på följande sätt, de inringade delarna är sinsemellan likformiga.

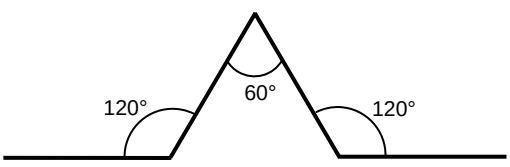


Fraktala kurvor kan definieras iterativt eller rekursivt. I fallet ovan har fyra iterationer använts. När antalet iterationer går mot oändligheten så kommer de inringande områdena efter omskalning till samma storlek visa sig ha exakt samma form. [2]

Koch-flingan är uppbyggd av tre likadana fraktaler ordnade i en liksidig triangel, flingans *stomme*. För att definiera en sådan fraktal rekursivt konstruerar man först en *generator* som beskriver fraktalens grundform, oftast ett antal sammanhängande linjesegment. Därefter definieras fraktalen rekursivt enligt följande mall:

1. En *fraktal* av grad 0 är en rät linje.
2. En *fraktal* av grad *N* fås genom att ersätta varje linjesegment i en *fraktal* av grad *N*-1 med en skalad version av generatorn.

Generatorn för Koch-fraktalen ser ut så här:



[2] Faktum är att kurvan är oändlig men omsluter trots det en ändlig yta.

De fyra linjesegmenten är lika långa och utgör en tredjedel av hela generatorns bredd. Generatorn har alltså form men ingen specifik storlek, den skalas till samma storlek som linjesegmentet den ersätter i punkt 2 ovan. Följande figur visar Koch-fraktaler av grad 0, 1 och 2.

Sköldpaddsgrafik

Fraktala kurvor ritas enklast med sköldpaddsgrafik, som kan liknas vid att styra en sköldpaddas väg genom sanden. Sköldpaddan kan vrida sig i en viss riktning och gå en viss sträcka i aktuell riktning. När den går lämnar den ett spår (en linje) efter sig. Sköldpaddsgrafik lämpar sig väl för fraktalritning eftersom man slipper att räkna med koordinater. Här följer en enkel javaklass (utan detaljer):

```
public class Turtle {
      // draw a line in the current direction, distance long
    public void walk(double distance)

      // jump distance long in the current direction without drawing
    public void jump(double distance)

      // jump to point p
    public void jumpTo(Point p)

      // turn the drawing direction angle degrees
    public void turn(double angle)

      // change the drawing direction to absolute angle
    public void turnTo(double angle)
}
```

Vi kommer att definiera fraktala snöflingor som klasser. Den abstrakta klassen

```
public abstract class Flake {
    protected Turtle turtle = null; // initiated in subclasses
    public abstract void draw(Turtle turtle, int level, double size);
}
```

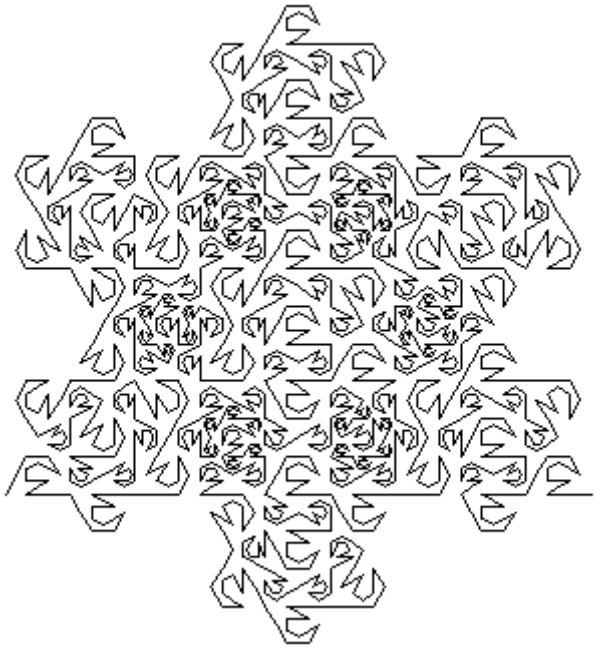
blir utgångspunkt för ett antal fraktalklasser. Här är Kochflingans klass:

```
public class ConvexKoch extends Flake {

    public void draw(Turtle turtle, int n, double size) {
        this.turtle = turtle;
        turtle.turnTo(0.0);
        // Draw a triangular structure of three fractals
        turtle.turn(-60.0);
        drawSide(n,size);
        turtle.turn(120.0);
        drawSide(n,size);
        turtle.turn(120.0);
        drawSide(n,size);
    }
```

forts.

```
    private void drawSide(int n,double size) {
        if ( n == 0 )
            turtle.walk(size);   // dra en rät linje
        else {
            double l = size/3.0;
            drawSide(n-1,l);
            turtle.turn(-60.0);
            drawSide(n-1,l);
            turtle.turn(120);
            drawSide(n-1,l);
            turtle.turn(-60);
            drawSide(n-1,l);
        }
    }
}
```

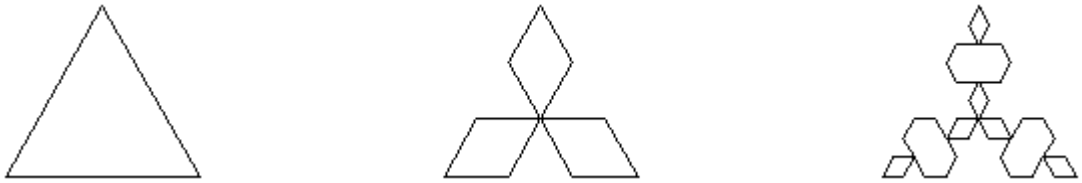
I mappen `fraktaler` finns färdiga klasser och ett GUI som kan användas som testbänk för olika fraktalklasser. Uppgifterna är nu att konstruera ett par andra fraktaler med utgångspunkt från exemplen ovan. När du definierat en ny subklass till `Flake` så är det bara att addera ett objekt av den till mappen i klassen `FlakeDB:s konstruktor`, så dyker det automatiskt upp en ny inmatningsruta för fraktalens grad i GUI:t. Den färdiga koden innehåller Koch-flingan ovan, samt Benoit Mandelbrots PS-kurva, en lite mer komplicerad kurva med fraktal dimension 2, vilket innebär att den fyller hela ytan när antalet iterationer går mot oändligheten (testa med N = 5). Här efter 3 iterationer:



(*Om du råkar skriva in för stora tal tar beräkningen för lång tid. Gå då tillbaks till verktyget och avbryt exekveringen, det kan tyvärr inte göras i nuvarande version av GUI:t*)

Uppgift C.1 Konkav Koch-flinga

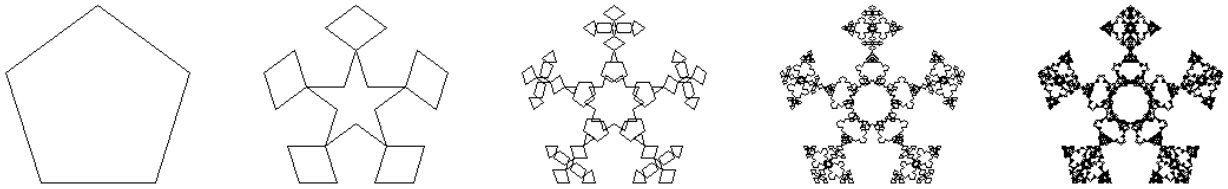
Definiera klassen `ConcaveKoch`. De tre första iterationerna ser ut så här:



Tips: Denna flinga har samma stomme och generator som Koch-flingan vi sett tidigare. Det är faktiskt bara en liten detalj som skiljer.

Uppgift C.2 Penta

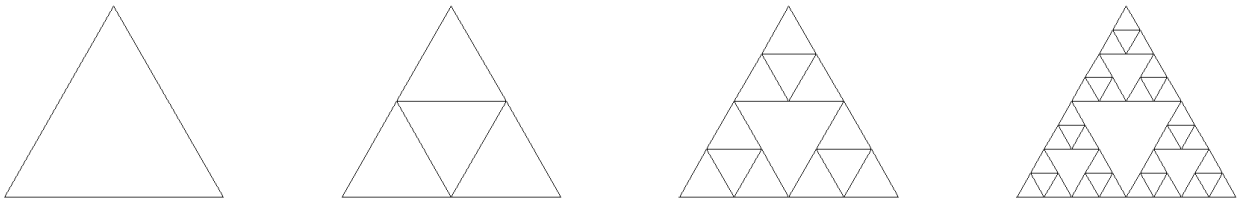
Flingans fem första iterationer:



Stommen är en liksidig pentagon. Identifiera generatorn genom att studera figurerna och definiera klassen `Penta`! *Tips*: Denna kurva korsar sig själv vilket bidrar till att göra bilderna ovan lite mer svårtolkade än vanligt.

Uppgift C.3 Sierpinskitriangel

Flingans fyra första iterationer:



Generatorn är här inte en linje utan en liksidig triangel. I varje steg ersätts generatorn med tre förminskade versioner av generatorn.

Penta efter fyra iterationer