Laboration 4.

Syfte

Syftet med denna laboration är att få övning i att strukturera sina program genom att använda metoder och klasser, samt att få övning i att använda sig av fält och klassen ArrayList.

Redovisning

Uppgifterna skall vara demonstrerade för och godkända av en handledare senast måndag 23/2.

Uppgift 1

Skriv en klassmetod

public static boolean match(String str)

som kontrollerar parentesmatchning i strängar. Metoden tar en sträng str som indata och returnerar värdet **true** eller **false** beroende på om parenteserna i strängen str matchar eller inte.

Exempel:

I exemplen nedan representerar . . en följd av godtyckliga tecken som inte innehåller vänster- eller högerparenteser:

. .(. .(. .). .).	skall ge värdet **true**
. .(. .(. . .).	skall ge värdet **false**
. .(. . . .). .(. . . .). .	skall ge värdet **true**
. . . .). .). .(. .(. . . .	skall ge värdet **false**
. .(. .). .). .(. .(. .). .	skall ge värdet **false**
. . . . .	skall ge värdet **true**

Givetvis skall du också skriva ett program för att testa metoden.

Tips: Räkna antalet förekomster av vänster- respektive högerparenteser under genomlöpningen av strängen. Vad måste gälla för förhållande mellan dessa värden - *vid varje tidpunkt under genomlöpning och efter avslutad genomlöpning* - för att parentesmatchningen skall vara korrekt?

Uppgift 2

I denna uppgift skall ni använda klassen GenericDie som ni gjorde i laboration 3.

Ni skall skapa en klass TestDice som innehåller en klassmetod

public static int[] roll(GenericDie theDice, **int** nrRoll)

som får en tärning theDice av klassen GenericDie, kastar denna tärning nrRoll antal gånger och returnerar ett heltalsfält som innehåller hur många gånger var och en av valörerna på tärningen kommit upp.

Skriv också en main-metod som använder metoden roll för att göra 100 kast med en 7-sidig tärning och skriver ut resultatet av dessa kast.

Tips: Fältet som metoden roll returnerar innehåller lika många element som det finns sidor på tärningen theDice. Klassen GenericDie har en metod för att ta reda på antalet sidor på tärningen.

Valörerna på en tärning är numrerade från 1 till max, medan indexen i ett fält är indexerade från 0 till max-1. Var i fältet är det lämpligt att lagra valören 1, valören 2, . . , valören max?

Uppgift 3

Utöka klassen TestDice i föregående uppgift med ytterligare en klassmetod

public static int[] valueOf(**int**[] number)

som tar ett heltalsfält number innehållande antal gånger som var och en av valörerna på en tärning kommit upp. Metoden skall returnera ett nytt fält som för varje valör innehåller den *sammanlagda poängsumman* för antalet gånger valören kommit upp. Har t.ex. valören 6 kommit upp 4 gånger skall värdet i returfältet som associeras med valören 6 vara 24.

Skriv sedan en main-metod som gör 100 kast med en 9-sidig tärning samt skriver ut antal förekomster av och den totala poängsumman för varje valör, d.v.s. programmet skall först anropa metoden roll (från förra uppgiften) för att generera antalet förekomster av varje valör och därefter metoden valueOf för att beräkna de totala poängsummorna.

Tips: Fältet som metoden valueOf returnerar innehåller lika många element som indatafältet number. Hur kan man ta reda på storleken på ett fält?

Uppgift 4

A är en kvadratisk heltalsmatris med N rader och N kolonner. I första kolonnen i A är alla elementen lika med 1. I första raden i A är alla övriga element lika med 0. För resterande element i A gäller att

$$a_{i\,j} = a_{i-1\,j-1} + a_{i-1\,j}$$

I fallet N = 5 får matrisen utseendet:

$$A = \begin{matrix} 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 \\ 1 & 2 & 1 & 0 & 0 \\ 1 & 3 & 3 & 1 & 0 \\ 1 & 4 & 6 & 4 & 1 \end{matrix}$$

Skriv ett program som läser in ett positivt heltal N ≤ 10, beräknar matrisen A, samt skriver ut alla element i och till vänster om diagonalen hos A enligt nedan:

```
1
1   1
1   2   1
1   3   3   1
1   4   6   4   1
```

Utforma programmet på så sätt att det förutom main-metoden innehåller följande tre klassmetoder

public static int readGrad()	som läser in och returnerar ett gradtal ≤ 10
public static int[][] createArray(**int** n)	som skapar och returnerar en matris av gradtal n
public static void writeElement(**int**[][] m)	som skriver ut elementen i matrisen m

Tips 1: Börja vid beräkningen av matrisen att initialisera alla element i första raden till 0 och sedan alla element i första kolumnen till 1.

Tips 2: Uppgift 13 i laboration 1 är till hjälp vid implementationen av metoden writeElement.

Tips 3: Utveckla och testa en metod i taget. Börja med den metod du tror är lättast att utveckla.

Uppgift 5

I laboration 3 skrev ni en klass Guest för att representera gäster på hälsoinstitutet Svett&Fett.

Skriv en klass Diet som innehåller klassmetoden

public static ArrayList<Guest> putOnDiet(ArrayList<Guest> list)

Metoden tar en lista list av Guest och returnerar en ny lista, som innehåller de element i list som är överviktiga.

Ni kan testa med den färdiga klassen TestDiet, som finns på kursens hemsida.

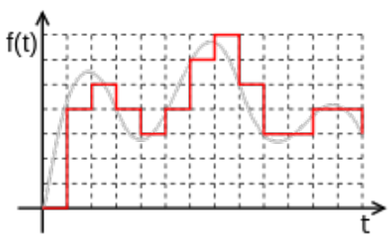
Uppgift 6

I denna uppgift ska ni vidareutveckla ett litet bibliotek för att generera musik som kan avlyssnas med vanliga musikprogram. Tiden medger inte något omfattande bibliotek, så den musik vi kan åstadkomma blir givitvis mycket begränsad.

En ljudsignal uppstår genom att en tryckvåg fortplantas genom luften, vilket är ett analogt fenomen. För att kunna göra datorbearbetningar måste vi ha en digital representation av den analoga ljudsignalen. Betrakta nedanstående figur:

Den grå analoga signalen har *samplats* (avlästs) vid diskreta tidpunkter (de vertikala streckade linjerna). Dessutom har signalvärdena (amplituderna) vid dessa tidpunkter avrundats till närmaste heltalsvärde, representerat av de horisontella streckade linjerna. Den samplade digitala signalen i figuren ovan är alltså

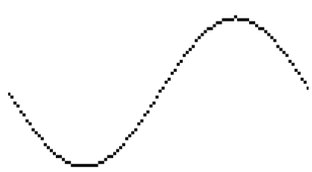
[0, 4, 5, 4, 3, 4, 6, 7, 5, 3, 3, 4, 4, 3].



En vanlig standard för digital musik, som t.ex. används för CD-skivor, har samplingsfrekvensen 44100 Hz (dvs 44100 avläsningar av signalen per sekund). Noggrannheten i amplitud är typiskt 16 bitar, dvs $2^{16} = 65536$ olika amplituder kan anges. I vårt bibliotek ska vi sampla signaler med denna frekvens, men representera amplituder med värden av typen **double**, skalade på så sätt att amplituden alltid ligger mellan -1 och 1. Omvandlingen till 16 bitarsrepresentation sker i klassen Melody som ni får given.

Det mänskliga örat kan uppfatta ljud ungefär i intervallet 20-20000 Hz. Grundfrekvensen för ettstrukna A (den ton efter vilken instrument stäms) är 440 Hz. När en sinussignal med denna frekvens samplas med samplingsfrekvens 44100 Hz erhåller man alltså ca 100 värden per period, dvs den samplade signalen ger en mycket god bild av den analoga signalen. Nedan visas en period av en sinuskurva, samplad med 100 punkter:

För högre frekvenser blir givetvis upplösningen sämre, men för hörbara signaler har man dock minst två avläsningar per period.



Att lagra digitala ljudsignaler är minneskrävande. Med 44100 avläsningar per sekund och 16 bitar per avläsning får vi för monoljud en minnesåtgång på 88 kbytes per sekund, dvs ca 5 Mbytes per minut. För stereoljud fördubblas minnesåtgången till 10 Mbytes per minut. För ljudfiler används därför ofta komprimerade format som MP3, vilka tappar en del ljudinformation men ger mycket mindre filer. Vi ska dock i denna laborationsuppgift hålla oss till okomprimerade filer i formatet WAV.

Förberedelser:

Skapa en ny mapp för denna laborationuppgift och ladda ner filen musicLab4.zip från kurshemsidan till denna mapp. Filen innehåller 4 klasser: SoundDevice, Melody, MusicUtils och Main, samt en textfil elise.txt.

Ladda in filen Main.java i JGrasp samt kompilera och kör programmet (med hörlurar på!). Ni hör en ren sinussignal med frekvensen 440 Hz i två sekunder, följt av en lika lång signal en oktav högre (dvs. med frekvensen 880 Hz).

Filen Main.java har följande utseende:

```
import java.io.File;
public class Main {
    public static void main(String[] args) {
        SoundDevice device = new SoundDevice();
        Melody song = new Melody(5);
        melody.add(MucicUtils.sine(440, 2));
        melody.add(MucicUtils.sine(880, 2));
        melody.play(device);
        melody.save(device.getFormat(), new File("twotones.wav"));
    }//main
} //Main
```

Programmet börjar med att skapa ett objekt av klassen SoundDevice och ett objekt av klassen Melody. Båda dessa klasser är givna och ni skall *inte ändra* i dessa.

Klassen SoundDevice innehåller operationer som gör det möjligt att komma åt och hantera datorns ljudkort. För laborationen behöver vi inte förstå hur klassen SoundDevice är uppbyggd. Det räcker att veta att konstanten SAMPLING_RATE = 44100 definieras i denna klass.

Klassen Melody används för att avbilda en melodi. Klassens konstruktor har en **int** som parameter. Denna parameter anger melodins längd i sekunder. I main-metoden skapas alltså en melodi som är 5 sekunder lång. Initialt är melodin ljudlös. Klassen Melody innehåller metoden add för att lägga till toner, metoden play för att spela melodin, samt metoden save för att spara melodin på en fil.

I main-metoden anropas metoden add två gånger för att lägga till toner i melodin. Tonerna som läggs till skapas med metoden sine, som finns i klassen MusicUtils. Metoden sine producerar en sinussignal med given frekvens och längd. Melodin spelas sedan upp genom att anropa metoden play. Slutligen spara main-metodens melodin på en fil med namnet "twotones.wav" genom att anropa metoden save. Filen twotones.wav kan öppnas och spelas upp med datorns CD-spelare. Gör detta och kolla att ni höra de två tonerna.

Metoden sine i klassen MusicUtils har följande utseende:

```
public static double[] sine(double freq, double duration) {
    int length = (int) (duration*SoundDevice.SAMPLING_RATE);
    double[] a = new double[length];
    double dx = 2*Math.PI*freq / SoundDevice.SAMPLING_RATE;
    for (int i = 0; i < length; i = i + 1) {
        a[i] = Math.sin(i * dx);
    }
    return a;
}//sine
```

Metoden skapar och fyller ett fält a med avläsningar av en ren sinusfunktion. Antalet värden i fältet är antalet avläsningar per sekund (som definieras av konstanten SoundDevice.SAMPLING_RATE) gånger tonens längd i sekunder (som anges av inparametern duration).

Den funktion vi ska sampla är $s(t) = \sin(2\pi f \cdot t)$, där f är frekvensen (som anges av inparametern freq).

Vi fyllar fälet a med SoundDevice.SAMPLING_RATE avläsningar per sekund av funktionen s(t), således får fältelementet a[i] värdet s(i / SoundDevice.SAMPLING_RATE).

Varför har vi i koden för metoden sine, ovan, infört variabeln dx?

a) För att skapa ett musikaliskt intressantare ljud än en ren sinuston skall ni utöka klassen MusicUtility med en metod

public static double[] pluck(**double** freq, **double** duration)

Metoden skapar och fyller ett fält a med avläsningar av en ren sinusfunktion. Antalet värden i fältet är antalet avläsningar per sekund (som definieras av konstanten SoundDevice.SAMPLING_RATE) gånger tonens längd i sekunder (som anges av inparametern duration).

Karplus-Strongs algoritm fungerar på följande sätt:

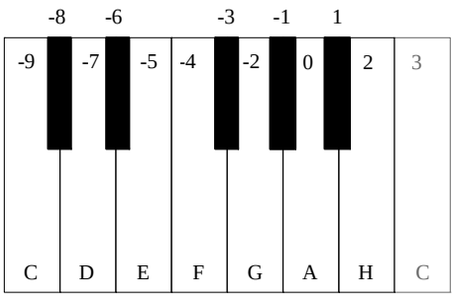
Först måste ni deklarera och skapa ett fält av flyttal av lämplig storlek på precis samma sätt som i funktionen sine.

Sedan ska ni fylla ni fältets element med värden. Detta görs i två steg:

1. Låt p vara antalet avläsningar per period, dvs samplingsfrekvensen 44100 dividerat med frekvensen freq för den önskade tonen. Fyll de första p elementen med slumptal i intervallet [-1.0, 1.0[. För att få slumptal använder instansmetoden nextDouble() i klassen Random. Om ni tittar i API'n, ser ni att metoden nextDouble() ger ett ett värde i intervallet [0.0, 1.0[. Hur gör man om detta värde till ett tal mellan -1 och 1?
2. Övriga element i fältet, dvs. elementen efter de *p* första elementen, beräknas på följande sätt: elementet med index *i* är summan av elementen med index *i-p* och *i-(p-1)*, multiplicerad med en dämpkonstant *d*. Ett lämpligt värde på *d* är 0.498, men ni får gärna experimentera med olika värden (som ger olika ljud). Intressanta ljud fås bara för *d*-värden litet mindre än 0.5.

Testa er metod genom att ändra i main-metoden så att ni anropar pluck i stället för sine. Ändra också namnet på filen där ni sparar melodin. Ni bör fortfarande höra två toner, men nu ska de låta som när man knäpper på en sträng.

b) I metoderna sine och pluck anges toner genom att ge frekvens och varaktighet. Vi vill nu närma oss hur toner anges normalt i musik, dvs med noter i en skala. Vi kan inte gå igenom musikteori här, utan nöjer oss med att konstatera att det, till exempel på en pianoklaviatur, finns tolv tangenter per oktav, sju vita och fem svarta:



Vi skall inte ange toner med de vanliga bokstäverna CDEFGAH, eftersom det kräver förtecken för de svarta tonerna och dessutom något sätt att ange oktav. Istället numrerar vi tonerna med heltal som på klaviaturen ovan, där ettstrukna A ges nummer 0 (vi väljer A som utgångspunkt, eftersom den, med frekvensen 440 Hz, oftast är referenspunkt för musikaliska skalor). Ettstrukna C är alltså -9, tvåstrukna C är 3, trestrukna C är 15, osv. Detta har också fördelen att vi lätt får en direkt formel för frekvensen för en given ton: ton nummer *k* har frekvensen $440*2^{k/12}$ Hz. Ettstrukna C har alltså frekvensen $440*2^{-9/12} = 261.63$ Hz.

Er uppgift är nu att utöka MusicUtils med ytterligare en metod

public static double[] note(**int** pitch, **double** duration)

som genererar en ton med givet nummer (anges med parametern pitch) och varaktighet (anges med parametern duration). Observera: Det enda man behöver göra är att räkna ut frekvensen från tonens nummer, anropa `pluck` för att få ett fält och returnera detta fält. I metoden note ska man varken skapa eller fylla fältet; det sköter pluck om.

När ni gjort detta kan ni testa resultatet genom att ändra i main-metoden så att programmet spelar till exempel början på Gubben Noak:

```
melody.add(MusicUtils.note(-9, 0.4));
melody.add(MusicUtils.note(-9, 0.4));
melody.add(MusicUtils.note(-9, 0.4));
melody.add(MusicUtils.note(-5, 0.4));
melody.add(MusicUtils.note(-7, 0.4));
melody.add(MusicUtils.note(-7, 0.4));
melody.add(MusicUtils.note(-7, 0.4));
melody.add(MusicUtils.note(-4, 0.4));
melody.add(MusicUtils.note(-5, 0.4));
melody.add(MusicUtils.note(-5, 0.4));
melody.add(MusicUtils.note(-7, 0.4));
melody.add(MusicUtils.note(-7, 0.4));
melody.add(MusicUtils.note(-9, 1.0));
melody.play(device);
```

Valet av 0.4 sekunder per ton är en smaksak; ni kan välja annat tempo om ni så önskar. Vi ser också att det blir mycket omständligt att beskriva musik på detta sätt. Men innan vi ger oss på att åtgärda detta ska vi förbättra ljudet ytterligare.

c) Ett betydligt intressantare ljud fås om vi spelar flera toner samtidigt. Som ett första steg ska ni skriva en metod

public static double[] average(**double**[] t1, **double**[] t2)

som genererar ett medelvärde av tonerna t1 och t2. Vi förutsätter också att de två tonerna har samma varaktighet så att fälten är lika långa. Metoden average ska helt enkelt skapa ett nytt fält och för varje index ta medelvärdet av motsvarande värden i t1 och t2.

Nästa steg är att ni skall skriva en metod

public static double[] harmonic(**int** pitch, **double** duration)

som skapar en ton med givet nummer och varaktighet genom att blanda *tre* toner som skapats med note, nämligen tonerna med nummer pitch, pitch-12 och pitch+12 (dvs en grundton samt en ton en oktav under och en ton en oktav över). Förslagsvis blandas först de två sistnämnda tonerna med metoden average och sedan blandas resultatet med den första tonen, återigen med metoden average.

Slutligen skall ni testa resultatet genom att i main-metoden ersätta alla anrop till metoden note med anrop till metoden harmonic.

d) I main ovan är beskrivningen av Gubben Noak en del av programmet. Mycket bättre är att spara beskrivningen av melodin i en textfil noak.txt som börjar

```
-9  0.4
-9  0.4
-9  0.4
-5  0.4
-7  0.4
-7  0.4
-7  0.4
-4  0.4
-5  0.4
-5  0.4
-7  0.4
-7  0.4
-9  1.0
```

och sedan låta programmet läsa denna fil och med hjälp av informationen på varje rad skapa en ton som läggs till sången. Ännu bättre är kanske att ändra varaktigheten för tonerna i filen till 0.25, svarande mot en kvartston, och sedan separat ange tempot.

Ändra nu Main så att beskrivningen av en melodi läses från en fil genom att ge filnamnet som argument till main-metoden. Koppla sedan filen till ett Scanner-objekt. Detta görs på följande vis

```
public static void main(String[] args) throws FileNotFoundException {
        Scanner sc = new Scanner(new File(args[0]));
        . . .
}//main
```

Klassen Scanner har en konstruktor som göra att Scanner-objektet läser data från en fil. En fil handhas som ett objekt av klassen File och klassen File har en konstruktor vars parameter är namnet på den fysiska filen.

Vid anropet av **new** File(arg[0]) inträffar en exception av typen FileNotFoundException om den angivna fysiska filen inte finns. FileNotFoundException tillhör en typ av exceptions som måste tas hand om eller kastas vidare. Här kastar vi FileNotFoundException vidare, därav står det **throws** FileNotFoundException i metodhuvudet till main-metoden.

För att testa ert program kan ni använda den givna textfilen elise.txt. Denna fil utnyttjar ovan antydda förbättring; varaktigheten 0.125 betyder en åttondelston. Om man låter detta vara synonymt med varaktigheten hos tonen i sekunder så blir det alldeles för högt tempo. Om en åttondel varar i 0.125 sekunder så hinner man 240 fjärdedelsnoter på en minut; 148 fjärdedelar per minut är mer lagom. En möjlighet är att även ange tempot som argument till main-metoden. Gör det!

Ni kan naturligtvis också skriva en egen fil för någon musik ni föredrar.

Slutkommentarer:

Förhoppningsvis har ni nu lärt er en del programmering med fält och en del om digital audio. Avslutningsvis skall påpekas att vi i ett avseende gjort det enkelt för oss: vi har slösat med minne. Vi skapar många och långa fält också för korta ljudsekvenser. Det går att göra det vi gjort, och mer därtill, med mycket mindre minneskonsumtion. För ett allvarligt menat musikprogram måste man vara mer ekonomisk, men för vårt syfte här lämpar sig en mer frikostig attityd till minnesförbrukning bättre.

Uppgift 7

En digital bild kan representeras som ett tvådimensionellt fält av bildpunkter.

I en digital gråskalebild är en bildpunkt ett heltalsvärde i intervallet 0-255, där värdet 0 betecknar svart och värdet 255 betecknar vitt. En bild har en viss storlek, dvs ett visst antal bildpunkter i höjdled och ett visst antal bildpunkter i sidled. En gråskalebild kan således representeras med ett tvådimensionellt fält av typen **int**[][], där den första dimensionen definierar bildens höjd och den andra dimensionen definierar bildens bredd. En variabel som representera en gråskalebild med höjden HEIGHT pixlar och bredden WIDTH pixlar kan skapas med satsen

int[][] grayImage = **new int**[HEIGHT][WIDTH];

Elementet grayImage[20][40] anger således gråtonen i den pixel som återfinns 40 pixlar åt höger och 20 pixlar nedåt från bildens övre vänstra hörn. Observera att grayImage[0][0] är pixeln i övre vänstra hörnet i bilden och att grayImage[HEIGHT-1][WIDTH-1] är pixeln i nedre högra hörnet.

En digital färgbild lagras på RGB-format, där varje bildpunkt utgörs av *tre* heltalsvärden i intervallet 0-255. De enskilda värdena representerar intensiteten av färgerna rött, grönt och blått. En färgbild kan avbildas med ett tredimensionellt fält av typen **int**[][][], är den första dimensionen definierar bildens höjd, den andra dimensionen definierar bildens bredd och den tredje dimensionen representerar färgerna rött, grönt och blått. En variabel som representera en färgbild med höjden HEIGHT pixlar och bredden WIDTH pixlar kan skapas med satsen

int[][][] colorImage = **new int**[HEIGHT][WIDTH][3];

Elementet colorImage[18][56][1] anger således intensiteten av grönt i den pixel som återfinns 56 pixlar åt höger och 18 pixlar nedåt från bildens övre vänstra hörn.

I denna uppgift skall ni skriva några enkla metoder för bildbehandling. På kursens hemsida finns ett antal färdiga klasser som ni skall använda.

Gråskalebilder.

I denna uppgift skall ni skriva metoder för behandling av gråskalebilder. Ladda ner filen grayLab4.zip från kursens hemsida. Filen innehåller klasserna GrayImage, GrayImagePanel, GrayImageWindow , GrayUtils och MyGrayProgram, samt bildfilen svamp.jpeg.

Klassen GrayImage tillhandahåller de publika klassmetoderna

static int[][] read(String fileName) — som läser in en bild på gif -, jpg-, jpeg- eller png-format från filen fileName. Om ursprungsbilden i filen är en färgbild översätts denna till en gråskalebild.

static void write(String fileName, **int**[][] picture) — som skriver ut fältet picture som en gråskalebild på formatet gif, jpg, jpeg eller png till en fil med namnet fileName.

Klassen GrayImageWindow tillhandahåller en konstruktor

GrayImageWindow(**int**[][] picture1, **int**[][] picture2) — som skapar ett fönster i vilket fälten picture1 och picture2 ritas ut som gråskalebilder.

Klassen MyGrayScaleProgram innehåller nedanstående kod och producerar fönstret som visas ovan:

```
public class MyGrayProgram {
    public static void main(String[] args) throws Exception{
        int[][] original = GrayImage.read("svamp.jpeg");
        int[][] manipulated = GrayUtils.upDown(original);
        GrayImage.write("upDownSvamp.jpeg", manipulated);
        GrayImageWindow iw = new GrayImageWindow(original, manipulated);
    }
}//MyGrayProgram
```

I klassen GrayUtils finns metoden

```
public static int[][] upDown(int[][] samples) {
    int[][] newSamples = new int[samples.length][samples[0].length];
    for (int row = 0; row < samples.length; row = row + 1)
        for (int col = 0; col < samples[row].length; col = col + 1)
            newSamples[row][col] = samples[samples.length - row - 1 ][col];
    return newSamples;
}//upDown
```

som tar ett tvådimensionellt fält samples (som representerar en gråskalebild) och returnerar ett nytt tvådimensionellt fält (som också representerar en gråskalebild). Det nya fältet som metoden returnerar är en spegelvänd kopia av fältet samples roterad runt den horisontella axeln, dvs första raden i det nya fältet är sista raden i samples, andra raden i det nya fältet är näst sista raden i samples, osv. Metoden kan alltså användas för att vända en bild upp och ner.

Metoden main i klassen MyGrayScaleProgram läser in en bildfil (svamp.jpeg) som innehåller en gråskalebild och lagrar denna i det tvådimensionella fältet original, skapar ett nytt fält manipulated genom att anropa metoden upDown med fältet original, skriver ut fältet manipulated som en bild till filen upDownSvamp.jpeg och skapar slutligen ett fönster där filerna original och manipulated ritas ut som bilder.

Din uppgift är att utöka klassen GrayUtils med följande metoder:

public static int[][] leftRigt(**int**[][] samples)	som returnerar en spegelvänd kopia av samples roterad runt den vertikala axeln.
public static int[][] invert(**int**[][] samples)	som returnerar en kopia av samples där värdet $s_{i,j}$ för varje element i samples har ersätts med värdet 255- $s_{i,j}$.
public static int[][] toBlackWhite(**int**[][] samples)	som returnerar en kopia av samples där värde $s_{i,j}$ för varje element i samples har ersätts med värdet 0 om $s_{i,j}$ är mindre än 128 och med värdet 255 om $s_{i,j}$ är större eller lika med 128.


orginal


leftRight


invert


toBlackWhite

För att rita ut t.ex. den spegelvända kopian ändrar ni satsen

```
int[][] manipulated = GrayUtils.upDown(original);
```

till

```
int[][] manipulated = GrayUtils.leftRight(original);
```

i main-metoden i klassen MyGrayScaleProgram. Vill ni lagra den nya bilden som en jpeg-bild ändrar ni också filnamnet i satsen

```
GrayImage.write("upDownSvamp.jpeg", manipulated);
```

Färgbilder:

För att handha färgbilder laddar ni ner filen colorLab4.zip från kursens hemsida. Filen innehåller klasserna ColorImage, ColorImagePanel, ColorImageWindow, ColorUtils och MyColorProgram bildfilen svamp.jpeg.

Klassen ColorImage tillhandahåller klassmetoderna

static int[][][] read (String fileName)	som läser in en bild på gif -, jpg-, jpeg- eller png-format från filen fileName.
static void write(String fileName, **int**[][][] picture)	som skriver ut fältet picture som en bild på formatet gif, jpg, jpeg eller png till en fil med namnet fileName.

Klassen ColorImageWindow tillhandahåller en konstruktor

ColoImageWindow(**int**[][][] picture1, **int**[][][] picture2)	som skapar ett fönster i vilket fälten picture1 och picture2 ritas ut som färgbilder.

Klassen MyColorProgram innehåller nedanstående kod och producerar fönstret som visas i bredvidstående bild:

```
public class MyColorProgram {
    public static void main(String[] args) throws Exception {
        int[][][] original = ColorImage.read("svamp.jpeg");
        int[][][] manipulated = upDown(original);
        ColorImage.write("upDownSvamp.jpeg", manipulated);
        ColorImageWindow iw = new ColorImageWindow(original, manipulated);
    }//main
}// MyColorProgram
```

I klassen ColorUtils finns metoden

```
public static int[][][] upDown(int[][][] samples) {
    int[][][] newSamples = new int[samples.length][samples[0].length][3];
    for (int row = 0; row < samples.length; row = row + 1)
        for (int col = 0; col < samples[row].length; col = col + 1)
            for (int c = 0; c < samples[0][0].length; c = c + 1)
                newSamples[row][col][c] = samples[samples.length-row-1][col][c];
    return newSamples;
}//upDown
```

som tar ett tredimensionellt fält samples (som representerar en färgbild) och returnerar ett nytt tredimensionellt fält (som också representerar en färgbild). Det nya fältet som metoden returnerar är en spegelvänd kopia av fältet samples roterad runt den horisontella axeln, dvs första raden i det nya fältet är sista raden i samples, andra raden i det nya fältet är näst sista raden i samples, osv. Metoden kan alltså användas för att vända en bild upp och ner.

Din uppgift är att utöka klassen ColorUtils med följande metoder:

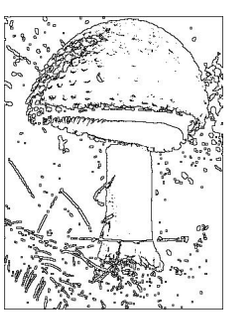
public static int[][][] leftRight(**int**[][][] samples)	som returnerar en spegelvänd kopia av samples roterad runt den vertikala axeln.
public static int[][][] invert(**int**[][][] samples)	som returnerar inversen av samples
public static int[][][] toGray(**int**[][][] samples)	som returnerar en gråbildskopia av samples
public static int[][][] toBlackWhite(**int**[][][] samples)	som returnerar en svartvit kopia av samples
public static int[][][] contour(**int**[][][] samples)	som returnerar konturen av samples


leftRight


invert


toGray


toBlackWhite


contour

Inversen till en färgbild erhålls genom att för varje pixel i orginalbilden ersätta intensiteten $s_{i,j}$ *för var och en av färgerna* röd, grön och blå med värdet 255- $s_{i,j}$.

För att få bästa kvalitén på den gråskalebild (och därmed den svartvita bild) som erhålls från en färgbild skall ni beräkna bildpunkternas *luminans*. Luminansen L är den för ögat upplevda ljusheten hos en yta och definieras som

$$L = 0.299\,r + 0.587\,g + 0.114\,b.$$

där r är intensiteten av rött, g är intensiteten av grönt och b är intensiteten av blått. I den gråskalebild som beräknas från en färgbild, sätts intensiteten av färgerna rött, grön och blått i varje pixel till värdet L i motsvarande pixel i färgbilden. I den svartvita bild som erhålls från en färgbild, sätt intensiteten av färgerna rött, grön och blått i varje pixel till värdet 0 om L i motsvarande pixel i färgbilden är mindre än 128 och till 255 om L är större eller lika med 128.

En *kontur* i en svartvit bild kan definieras enligt nedan:

> I konturen av bilden picture, skall en pixel vara svart om och endast om picture[i][j] är svart och antingen (i, j) ligger på *randen* i picture eller (i, j) har minst en *granne* som är vit i picture. Randen i bilden utgörs av de yttre raderna och kolumnerna. Grannar till (i, j) är alla (i_k, j_k) sådan att $i_k \in$ i-1..i+1 och $j_k \in$ j-1..j+1.

Bildpunkter som ligger på randen kan inte direkt beräknas med detta uttryck eftersom dessa bildpunkter saknar en eller flera närliggande punkter. Ni kan bortse från bildpunkterna på randen och låta dessa ha samma värden som i den svartvita varianten av originalbilden.

Frivilliga extrauppgifter

I bildanalys används ofta så kallade *faltningsfilter* för att lyfta fram sådant som är intressant i bilden. Det kan t.ex. röra sig om att ta bort brus, eller att reducera eller framhäva konturer i bilden. Vid användning av faltningsfilter beräknas ett nytt värde på en bildpunkt genom att, förutom att beakta bildpunkten själv, även beakta närliggande bildpunkter. Ett faltningsfilter kan således anses som en matris. Faltningsfiltret

-1	-1	-1
-1	9	-1
-1	-1	-1

är ett filter som *förändrar skärpan* i bilden. Filtret anger att det nya värdet i en bildpunkt beräknas genom att multiplicera bildpunktens gamla värde med 9 och subtrahera med varje närliggande bildpunkt. Detta kan också uttryckas på följande sätt:

> Alla bildpunkter $new_{i,j}$ i den bild som erhålls med faltningsfiltret och som *inte ligger på randen* i bilden beräknas av uttrycket
>
> $$new_{ij} = -1*old_{i-1,j-1} -1*old_{i-1,j} -1*old_{i-1,j+1} -1*old_{i,j-1} + 9*old_{i,j} -1*old_{i,j+1} -1*old_{i+1,j-1} -1*old_{i+1,j} -1*old_{i+1,j+1}$$
>
> där $old_{i,j}$ är värdet för bildpunkten i,j i bilden som filtreras. I en färgbild appliceras faltningsfiltret på varje färgkomponent. Värdet av den nya bildpunkten $new_{i,j}$ kan bli negativt eller större än 255. Detta måste kontrolleras. Negativa värden sätts till 0 och värden större än 255 sätts till 255.

Ett annat faltningsfilter som förändrar skärpan är

0	-1	0
-1	5	-1
0	-1	0

För att beräkna det nya värdet av en bildpunkt används i detta filter endast 4 av de 8 närbelägna bildpunkterna. Resultatet av de båda filtren ser ni nedan.

För att detektera kanter i bilder appliceras två filter på original bilden (ett filter för att få fram kanterna i x-led och ett filter för att få fram kanterna i y-led). Ett välkänt kantdekteringsfilter är Sobel-filtret, vars faltningsmatriser har följande utseende:

-1	-2	-1	1	0	-1
0	0	0	2	0	-2
1	2	1	1	0	-1

Om vi för varje enskild färgkomponent betecknar resultatet från dessa båda filter för bildpunkten (i, j) som $d_x(i, j)$ respektive $d_y(i, j)$, beräknas slutresultatet från Sobel-filteringen som $s(i,j)=\sqrt{d_x(i,j)^2+d_y(i,j)^2}$.

Färgbilden har först översatts till en gråskalebild, varefter gråskalebilden filtrerats med Sobel-filtren

Varje enskild färg i färgbilden har filtrerats med Sobel-filtren, varefter det erhållna värdet inverterats (dvs satts till 255 – värde).

Antalet bearbetningar man kan göra på bilder är näst intill obegränsat. Ni kan säkert komma på egna förslag på metoder att implementera. Den intresserade kan också hitta många ytterligare exempel på bildbehandling och filter genom sökning på nätet. Några lämpliga adresser att börja med är:

http://en.wikipedia.org/wiki/Image_editing

http://en.wikipedia.org/wiki/Sobel_operator