Datorlaborationer

Datorlaborationerna ger exempel på tillämpningar av det material som behandlas under kursen och ger träning i programutveckling. I detta kapitel finns alla laborationsuppgifterna. Schema (tider och datorsalar) finns i kursprogrammet.

Det finns 10 laborationer — 4 under vecka 4–7 av läsperiod ht1, 6 under vecka 1–6 av läsperiod ht2. Du måste redovisa varje laboration under ”rätt” vecka (om du inte är sjuk, se nedan). Om du är mycket ambitiös kan du dock arbeta i förväg och få flera uppgifter godkända vid ett redovisningstillfälle, under förutsättning att laborationsledaren har tid med detta. Du får däremot inte komma i efterhand och kräva att bli godkänd på flera uppgifter (utom om du har varit sjuk).

Regler för laborationerna:

- Laborationsuppgifterna ska lösas individuellt. Regler för samarbete finns i bilaga A, sidan 69.
- Till varje laborationsuppgift finns det läsanvisningar och förberedelseuppgifter. Före varje laboration måste du ha:
  - studerat läroboken enligt läsanvisningarna,
  - läst igenom *hela* laborationen noggrant,
  - löst förberedelseuppgifterna. I dessa uppgifter ska du skriva delar av de program som ingår i laborationen. Det krävs inte att allt du skrivit är helt korrekt, men du måste ha gjort ett rimligt försök. Kontakta en lärare om du får problem med uppgifterna.

  Vi förutsätter att du har förberett dig enligt ovan innan du kommer till laborationen — om du inte har gjort det så kommer du inte att få hjälp av laborationsledaren.

  Om du har förberett dig väl bör du hinna med alla uppgifterna under laborationen. Om du inte gör det så får du göra de resterande uppgifterna på egen hand och redovisa dem vid påföljande laborationstillfälle (och förbereda dig mera till nästa laboration).
- Om du är sjuk vid något laborationstillfälle så måste du anmäla detta till kursansvarig (`Per.Holm@cs.lth.se`, 046–222 80 38) före laborationen. Om du uteblir utan att ha anmält sjukdom så får du inte göra uppgiften förrän kursen går nästa gång, dvs nästa år, och då får du inte något slutbetyg i kursen i år. Om du varit sjuk bör du göra uppgiften på egen hand och redovisa den vid ett senare tillfälle. Det kommer också att anordnas en ”uppsamlingslaboration” i slutet av kursen.
- Du måste själv se till att laborationsledaren noterar dig som godkänd på varje uppgift (se godkännandebladet som finns sist i detta kompendium, sidan 83).

Laboration 1 — introduktion

Mål: Du ska lära dig vad ett datorprogram är och se exempel på ett enkelt program. Du ska också lära dig grunderna i användning av programutvecklingssystemet Eclipse.

Läsanvisningar

- Kapitel 1 i läroboken.
- Avsnitt 1–3 i bilaga C om Eclipse (sidan 77). Du kommer att använda Eclipse under hela kursen och även i kommande kurser.

Förberedelseuppgifter

Det finns inga förberedelseuppgifter till denna laboration.

Bakgrund

Ett datorprogram är ett antal rader text som beskriver lösningen till ett problem. Vi börjar med att titta på ett mycket enkelt problem: att från datorns tangentbord läsa in två tal och beräkna och skriva ut summan av talen. Lösningen kan se ut så här i Java:

```
1  package lab1;
2  import java.util.Scanner;
3
4  public class Calculator {
5      public static void main(String[] args) {
6          System.out.println("Skriv två tal");
7          Scanner scan = new Scanner(System.in);
8          double nbr1 = scan.nextDouble();
9          double nbr2 = scan.nextDouble();
10         scan.close();
11         double sum = nbr1 + nbr2;
12         System.out.println("Summan av talen är " + sum);
13     }
14 }
```

Detta är nästan samma program som beskrivs i Eclipse-handledningen, avsnitt 3, men talen som summeras läses från tangentbordet. Du behöver inte förstå detaljerna i programmet nu, men vi vill redan nu visa ett program som gör någonting intressantare än bara skriver ut ”Hello, world!”. Allt som behövs för att du ska förstå programmet kommer vi att gå igenom under de första veckorna av kursen. Förklaring av de viktigaste delarna av programmet:

- Rad 1: programmet ligger i ett ”paket” som heter `lab1`. Paket används för att samla program som hör ihop.
- Rad 2: en inläsningsklass `Scanner` importeras från paketet `java.util`. Detta paket och många andra ingår i Javas standardbibliotek.
- Rad 4: `public class Calculator` talar om att programmet, klassen, heter `Calculator`.
- Rad 5: `public static void main(String[] args)` talar om att det som vi skriver är ett ”huvudprogram”. Varje Javaprogram måste innehålla en klass med en `main`-metod. Exekveringen börjar och slutar i `main`-metoden.
- Rad 6: texten `Skriv två tal` skrivs ut på skärmen (`println` betyder ”print line”). När man använder Eclipse så hamnar utskrifter i konsolfönstret.

- Rad 7: inläsningsobjektet `scan` skapas. Vi kommer senare att gå igenom ordentligt vad detta betyder — nu räcker det att du vet att man alltid måste skapa ett sådant objekt när man vill läsa från tangentbordet.
- Rad 8: en variabel `nbr1` som kan innehålla ett reellt tal (`double`-tal) deklareras. Det betyder att man skapar plats för variabeln i datorns minne. Man tilldelar också `nbr1` ett talvärde som man läser in från tangentbordet med `nextDouble()`.
- Rad 9: samma sak med variabeln `nbr2`.
- Rad 10: scannern stängs när man är klar med inläsningen. Det är inte helt nödvändigt att stänga en scanner, men man får en varning om man inte gör det.
- Rad 11: variabeln `sum` deklareras. Talen `nbr1` och `nbr2` adderas och summan lagras i `sum`.
- Rad 12: resultatet (innehållet i variabeln `sum`) skrivs ut.
- Rad 13: slut på `main`-metoden.
- Rad 14: slut på klassen.

Uppgifter

1. Logga in på datorn. Öppna ett terminalfönster.

2. Under laborationerna kommer du att använda en hel del färdigskrivna eller nästan färdigskrivna program. Nu ska du skapa ett Eclipse-arbetsområde (workspace) som innehåller alla dessa filer.

   1. Ett förberett arbetsområde finns i packad form i filen *ptdc-workspace-14.zip* som finns i katalogen */usr/local/cs/ptdc/*. Flytta musmarkören till terminalfönstret så att det blir aktivt och skriv följande kommandon:

      ```
      cd
      unzip /usr/local/cs/ptdc/ptdc-workspace-14.zip
      ```

      `cd` betyder att du flyttar dig till din hemkatalog, `unzip` packar upp filen.

   2. Nu har du fått en katalog *ptdc-workspace* i din hemkatalog. Katalogen innehåller underkatalogerna *cs_eda016*, *cs_eda016_src*, *lab1*, *lab2*, *lab3*, ... Kontrollera innehållet i *ptdc-workspace* med följande kommando:

      ```
      ls ptdc-workspace
      ```

      I *ptdc-workspace* finns också en katalog *.metadata*. När du senare startar Eclipse skapas i din hemkatalog en katalog *.eclipse*. Dessa kataloger syns inte när du gör `ls`, eftersom deras namn inleds med punkt. Rör *inte* dessa kataloger — de används av Eclipse för att spara viktig information, och om de inte finns eller har fel innehåll så går det inte att starta Eclipse.

   Alternativt kan du hämta filen *ptdc-workspace-14.zip* från kursens hemsida:

   1. Starta en webbläsare och gå till kursens hemsida, `cs.lth.se/EDA016`.
   2. Klicka på Datorlaborationer.
   3. Ladda ner filen *ptdc-workspace-14.zip* till din hemkatalog.
   4. Packa upp filen enligt punkt 1 ovan.
   5. *.zip*-filen behövs inte mera. Tag bort den med följande kommando:

      ```
      rm ptdc-workspace-14.zip
      ```

3. Nu ska du starta Eclipse. Ge följande kommando:

   ```
   eclipse &
   ```

   Efter en stund får du en dialogruta där Eclipse frågar efter vilket arbetsområde som du vill använda. Ändra förslaget `workspace` till `ptdc-workspace`.

   `&`-tecknet betyder att Eclipse ska startas ”i bakgrunden”, alltså som ett fristående program. Det medför att man inte låser terminalfönstret utan kan arbeta med andra saker i det samtidigt som Eclipse kör. Alternativt kan du starta Eclipse genom att välja Eclipse (den senaste versionen) från menyn Applications > Programming.

   I Eclipse-fönstret finns i projektvyn (”Package Explorer”, längst till vänster) projekten *cs_eda016*, *cs_eda016_src*, *lab1*, *lab2*, *lab3*, ... — det är de projektkataloger som finns i *ptdc-workspace*. Projekten med namn som börjar på *lab* innehåller färdigskrivna program som utnyttjas under laborationerna. *cs_eda016* innehåller en biblioteksfil (*.jar*-fil) med klasser som utnyttjas. *cs_eda016_src* innehåller källkoden (*.java*-filerna) till dessa klasser — de behövs inte för laborationerna, men en del brukar vara intresserade av att titta på dem.

4. Du ska nu ladda in filen *Calculator.java* i en editor så att du kan ändra den. Man måste klicka en hel del för att öppna en fil:

   a) Öppna projektet *lab1* genom att klicka på pilen bredvid projektet.

   b) Öppna katalogen *src* genom att klicka på pilen.

   c) Öppna paketet *lab1* genom att klicka på pilen.

   d) Öppna filen *Calculator.java* genom att dubbelklicka på filnamnet. Filen öppnas i en editorflik.

5. Kör programmet: markera *Calculator.java* i projektvyn, klicka på Run-ikonen i verktygsraden (eller högerklicka och välj Run As > Java Application). I konsolfönstret skrivs texten `Skriv två tal`. Klicka i konsolfönstret, skriv två tal och tryck på RETURN. Observera: när man skriver reella tal ska man *vid inläsning* använda decimalkomma. När man skriver reella tal i program använder man decimalpunkt.

6. Ändra `main`-metoden i klassen `Calculator` så att fyra rader skrivs ut: talens summa, skillnad, produkt och kvot. Exempel på utskrift när talen 24 och 10 har lästs in:

   ```
   Summan av talen är 34.0
   Skillnaden mellan talen är 14.0
   Produkten av talen är 240.0
   Kvoten mellan talen är 2.4
   ```

   Du ska alltså efter utskriften av summan lägga in rader där talens skillnad, produkt och kvot beräknas och skrivs ut. Subtraktion anger man med tecknet `-`, multiplikation med `*`, division med `/`.

   Under tiden du skriver så kommer du att märka att Eclipse hela tiden kontrollerar så att allt du skrivit är korrekt. Fel markeras med kryss till vänster om raden. Om du håller musmarkören över ett kryss så visas en förklaring av felet. Om man sparar en fil som innehåller fel så visas felmeddelandena också i Problem-fliken längst ner.

   Du kommer också att märka att Eclipse kommer med förslag på vad du kan skriva. När du till exempel har skrivit `System.out.p` så får du upp en ruta med allt som man kan göra med `System.out` som börjar med `p`. Det här tycker en del är bra, andra tycker att det är irriterande. Under nästa laboration ska vi visa hur man stänger av denna hjälp, om man inte vill ha den.

7. Spara filen. Filen kompileras automatiskt när den sparas.

   När *.java*-filen kompileras skapas en ny fil med samma namn som klassen men med tillägget *.class*. Denna fil innehåller programmet översatt till byte-kod och används när programmet exekveras.

   Man ser inte *.class*-filerna inuti Eclipse, men de lagras i arbetsområdet precis som *.java*-filerna. *.java*-filerna finns i en katalog *src* under respektive projektkatalog, *.class*-filerna finns i en katalog *bin* (som inte syns i Eclipse).

8. Kör programmet och kontrollera att utskriften är korrekt. Rätta programmet om den inte är det.

9. Du ska nu använda Eclipse debugger för att följa exekveringen av programmet. Normalt använder man debuggern för att hitta fel i program, men här är avsikten bara att du ska se hur programmet exekverar rad för rad och att du ska bekanta dig med debuggerkommandona.

   - Sätt en brytpunkt på den första raden i `main`-metoden. Kör programmet under debuggern (Debug As > Java Application).
   - Svara ja på frågan om du vill byta till Debug-perspektivet.
   - Klicka på Step Over-ikonen några gånger. Notera att den rad i programmet som ska exekveras markeras i editorfönstret och att variabler som deklareras dyker upp i variabelvyn till höger. Variablernas aktuella värden anges också.
   - Sätt en brytpunkt på raden där du skriver kvoten mellan talen.
   - Klicka på Resume-ikonen för att köra programmet fram till brytpunkten.
   - Klicka på Resume igen för att köra programmet till slut.

   Byt tillbaka till Java-perspektivet genom att klicka på knappen Java längst till höger i verktygsfältet.

10. Följande program använder den färdigskrivna klassen `SimpleWindow`:

```
package lab1;
import se.lth.cs.window.SimpleWindow;

public class SimpleWindowExample {
    public static void main(String[] args) {
        SimpleWindow w = new SimpleWindow(500, 500, "Drawing Window");
        w.moveTo(100, 100);
        w.lineTo(150, 100);
    }
}
```

    Specifikationen av klassen `SimpleWindow` finns på kurshemsidan under Dokumentation och i appendix C i läroboken. Förklaring av de viktigaste delarna av programmet:

    - Raderna som börjar med `public` och de två sista raderna med `}` är till för att uppfylla Javas krav på hur ett program ska se ut.
    - På nästa rad deklareras en referensvariabel med namnet `w` och typen `SimpleWindow`. Därefter skapas ett `SimpleWindow`-objekt med storleken 500×500 pixlar och med titeln ”Drawing Window”. Variabeln `w` tilldelas det nya fönsterobjektet.
    - Sedan flyttas fönstrets ”penna” till punkten (100, 100).
    - Till sist ritas en linje till punkten (150, 100).

    Programmet finns inte i arbetsområdet, utan du ska skriva det från början. Skapa en fil *SimpleWindowExample.java*:

a) Markera projektet *lab1* i projektvyn,

b) Klicka på New Java Class-ikonen i verktygsraden.

c) Skriv namnet på klassen (`SimpleWindowExample`).

d) Klicka på Finish.

Filen laddas in i editorn automatiskt. Som du ser har Eclipse redan fyllt i `package`-raden, klassnamnet och de parenteser som alltid ska finnas. Komplettera klassen med `main`-metoden som visas ovan.

Spara filen, rätta eventuella fel och spara igen. Provkör programmet.

11. Ändra programmet genom att välja bland metoderna i klassen `SimpleWindow`. Till exempel kan du rita en kvadrat, ändra färg och linjebredd på pennan, rita fler linjer, skriva text, etc.

12. Om du har tid över: börja inte på laboration 2. Gör i stället följande:

Exekvera kalkylatorprogrammet och mata in talen `1 0` respektive `0 0`. Förklara resultatet (leta i läroboken, leta på webben, fråga laborationsledaren). Prova också att mata in ”talen” `two` och `four`. Förklara resultatet.

Ändra variablernas typ från reell (`double`) till heltal (`int`). Ändra också inläsningsmetoden från `nextDouble` till `nextInt`. Testa programmet och notera hur division fungerar när operanderna är heltal (detta kallas heltalsdivision, se avsnitt 6.3 i läroboken).

När du har ändrat variablernas typ till `int`: ändra programmet så att också värdet av uttrycket `nbr1 % nbr2` skrivs ut. Prova olika kombinationer av indata och försök sluta dig till vad `%`-operatorn gör. Leta sedan upp definitionen av `%`-operatorn i läroboken.

Laboration 2 — användning av färdigskriven klass

Mål: Du ska lära dig att läsa specifikationer av klasser och att utnyttja en färdigskriven klass för att lösa enkla uppgifter.

Läsanvisningar

- Kapitel 2 och avsnitt 3.1–3.2 i läroboken. Titta också på avsnitt 5.3 i boken.

Förberedelseuppgifter

- Gör uppgift 4 och 6.

Bakgrund

En klass `Square` som beskriver kvadrater har nedanstående specifikation. Detta är *inte* samma kvadratklass som beskrivs i läroboken — kvadratens läge är här koordinaterna för medelpunkten, inte övre vänstra hörnet, och kvadraten kan roteras kring sin medelpunkt.

```
/** Skapar en kvadrat med medelpunkten i x,y och med sidlängden side.
    Kvadratens sidor är parallella med koordinataxlarna */
Square(int x, int y, int side);

/** Tar reda på x-koordinaten */
int getX();

/** Tar reda på y-koordinaten */
int getY();

/** Tar reda på sidlängden */
int getSide();

/** Flyttar kvadraten avståndet dx i x-led, dy i y-led */
void move(int dx, int dy);

/** Ändrar sidlängden till newSide */
void setSide(int newSide);

/** Roterar kvadraten beta grader motsols kring sin medelpunkt */
void rotate(int beta);

/** Ritar kvadraten i fönstret w */
void draw(SimpleWindow w);

/** Raderar bilden av kvadraten från fönstret w. Kvadraten får
    inte flyttas mellan uppritning och radering */
void erase(SimpleWindow w);
```

I nedanstående program skapas först ett ritfönster. Därefter skapas en kvadrat som placeras mitt i fönstret och ritas upp.

```
package lab2;
import se.lth.cs.ptdc.square.Square;
import se.lth.cs.window.SimpleWindow;

public class DrawSquare {
    public static void main(String[] args) {
        SimpleWindow w = new SimpleWindow(600, 600, "DrawSquare");
```

```
        Square sq = new Square(300, 300, 200);
        sq.draw(w);
    }
}
```

Klassen finns i paketet `lab2`. I fortsättningen skriver vi inte ut paketen i programexemplen; som standard skapar Eclipse ett paket i varje projekt med samma namn som projektet.

Klasserna `SimpleWindow` och `Square` finns inte i Javas standardbibliotek utan har skrivits speciellt för denna kurs. Klasserna finns i paket vars namn börjar med `se.lth.cs`; det betyder att de är utvecklade vid institutionen för datavetenskap vid LTH, som använder domännamnet `cs.lth.se`.

Notera parametrarna som man använder när man skapar objekten. I `new`-uttrycket som skapar kvadratobjektet står det till exempel `(300,300,200)`. Det betyder att kvadraten har läget `300,300` (mitt i fönstret) och sidlängden `200`. Läget och sidlängden kan man ändra senare i programmet, med `sq.move(dx,dy)` och `sq.setSide(newSide)`.

I nedanstående program ritas en kvadrat många gånger. Efter varje uppritning minskas kvadratens sidlängd med `dim` pixlar. Uppritningen fortsätter så länge sidlängden är större än 0 pixlar. Läs igenom programmet och gör ditt bästa för att förstå det. `while`-satsen förklaras i läroboken avsnitt 3.2 och 7.5.

```
import java.util.Scanner;
import se.lth.cs.ptdc.square.Square;
import se.lth.cs.window.SimpleWindow;

public class DrawManySquares {
    public static void main(String[] args) {
        System.out.println("Skriv förminskning");
        Scanner scan = new Scanner(System.in);
        int dim = scan.nextInt();

        SimpleWindow w = new SimpleWindow(600, 600, "DrawManySquares");
        Square sq = new Square(300, 300, 200);
        while (sq.getSide() > 0) {
            sq.draw(w);
            sq.setSide(sq.getSide() - dim);
        }
    }
}
```

Observera att programmet ritar många bilder av samma kvadrat och att alla bilderna kommer att synas i fönstret när programmet är slut. Om man raderar den ”gamla” bilden innan man ritar en ny bild så kommer man att få en ”rörlig”, ”animerad”, bild. För att man ska se vad som händer måste man då göra en paus mellan uppritning och radering — det gör man med `SimpleWindow`-metoden `delay`, som har en parameter som anger hur många millisekunder man ska vänta. Exempel:

```
        while (sq.getSide() > 0) {
            sq.draw(w);
            SimpleWindow.delay(10);
            sq.erase(w);
            sq.setSide(sq.getSide() - dim);
        }
```

Anmärkning: i ett ”riktigt” program som visar rörliga bilder åstadkommer man inte animeringen på det här sättet. Denna lösning har bristen att man inte kan göra något annat under tiden som animeringen pågår, till exempel kan programmet inte reagera på att man klickar med musen.

Uppgifter

1. Logga in på datorn, starta Eclipse, öppna projektet *lab2*. (Logga in, starta Eclipse och öppna ett projekt ska du alltid göra, så det skriver vi inte ut i fortsättningen.)

2. Öppna en webbläsare och gå till kursens hemsida, `cs.lth.se/EDA016`. Under Dokumentation finns en länk till dokumentationen av de färdigskrivna klasser som används under laborationerna, alltså klasserna i paketet `se.lth.cs.ptdc`. Leta upp och titta igenom specifikationen av klassen `Square`. Det är samma specifikation som finns under Bakgrund, fast i något annorlunda form.

3. Klassen `DrawSquare` finns i filen *DrawSquare.java*. Öppna filen, kör programmet.

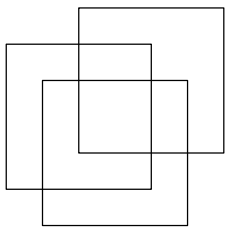
   Notera: i början av filen importeras två klasser, `SimpleWindow` och `Square`, men man ser bara den första `import`-satsen. Tryck på plustecknet till vänster om raden så visas alla importerna. Man kan dölja och visa metoder på samma sätt: prova med `main`-metoden.

4. Kopiera filen *DrawSquare.java* till en ny fil med namnet *DrawThreeSquares.java*. Enklast är att göra så här:

   1. Markera filen i projektvyn, högerklicka, välj Copy.
   2. Högerklicka på paketet *lab2*, välj Paste, skriv det nya namnet på filen.

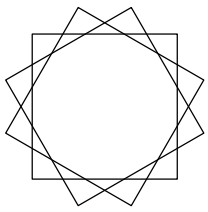
   Notera att Eclipse ändrar klassnamnet i den nya filen till `DrawThreeSquares`.

   Ändra sedan klassen så att kvadraten ritas tre gånger. Mellan uppritningarna ska kvadraten flyttas. Du ska fortfarande bara skapa ett kvadratobjekt i programmet. Resultatet ska bli en figur med ungefär följande utseende:



5. Testa programmet, rätta eventuella fel.

6. Kopiera filen till en ny fil *DrawThreeSquaresRot.java*. Ändra `main`-metoden så att en figur med nedanstående utseende ritas:



7. Klassen `DrawManySquares` finns i filen *DrawManySquares.java*. Kör programmet. Prova med olika värden på `dim` och studera resultatet. Övertyga dig om att du förstår hur programmet fungerar.

Ändra programmet så att kvadraten roteras `rot` grader efter varje uppritning (minskningen av storleken ska vara kvar). Läs in `rot` i början av programmet. Prova med olika värden på `rot` och `dim`.

8. Kopiera `DrawManySquares.java` till en ny fil `AnimatedSquare.java`. Lägg in fördröjning och radering så att kvadratbilden blir ”animerad”.

9. Programmet i uppgift 8 avslutas när kvadratens sidlängd blivit noll (eller mindre). Ändra programmet så att kvadraten i stället börjar växa när detta inträffar, och sedan börjar krympa när kvadraten blivit lika stor som fönstret, och sedan växa igen, och så vidare i all oändlighet.

Laboration 3 — händelsehantering

Mål: Du ska lära dig mera om att använda variabler och att skriva satser av olika slag. Dessutom kommer du att utnyttja operationer i `SimpleWindow` för att ta hand om musklick och tangenttryckningar. Till sist ska du lära dig att tolka felutskrifter under kompilering och exekvering och lära dig litet mera om Eclipse.

Läsanvisningar

- Avsnitt 6.6 och appendix C i läroboken.

Förberedelseuppgifter

- Gör uppgift 2, tänk igenom (skissa på papper) hur du ska lösa uppgift 3.

Bakgrund

Några av operationerna i `SimpleWindow` (`moveTo` för att flytta pennan och `LineTo` för att rita en linje) utnyttjas i operationen `draw` i klassen `Square`. `SimpleWindow` innehåller också operationer för att ta hand om musklick. Dessa operationer har följande beskrivning:

```
/** Väntar tills användaren har klickat på en musknapp */
void waitForMouseClick();

/** Tar reda på x-koordinaten för musens position vid senaste musklick */
int getMouseX();

/** Tar reda på y-koordinaten för musens position vid senaste musklick */
int getMouseY();
```

När exekveringen av ett program kommer fram till `waitForMouseClick` så ”stannar” programmet och fortsätter inte förrän man klickat med musen någonstans i programmets fönster. Ett program där man skapar ett fönster och skriver ut koordinaterna för varje punkt som användaren klickar på har följande utseende:

```
import se.lth.cs.window.SimpleWindow;

public class PrintClicks {
    public static void main(String[] args) {
        SimpleWindow w = new SimpleWindow(600, 600, "PrintClicks");
        while (true) {
            w.waitForMouseClick();
            System.out.println("x = " + w.getMouseX() + ", " + "y = "
                    + w.getMouseY());
        }
    }
}
```

`while (true)` betyder att repetitionen ska fortsätta ”i oändlighet”. Man avbryter programmet genom att välja Quit i File-menyn i `SimpleWindow`-fönstret.

Också i nedanstående program ska användaren klicka på olika ställen i fönstret. Nu är det inte koordinaterna för punkten som användaren klickar på som skrivs ut, utan i stället avståndet i x- och y-led mellan punkten och den förra punkten som användaren klickade på. Vi sparar hela tiden koordinaterna för den förra punkten (variablerna `oldX` och `oldY`).

```
import se.lth.cs.window.SimpleWindow;

public class PrintClickDists {
    public static void main(String[] args) {
        SimpleWindow w = new SimpleWindow(600, 600, "PrintClickDists");
        int oldX = 0; // x-koordinaten för "förra punkten"
        int oldY = 0; // y-koordinaten
        while (true) {
            w.waitForMouseClick();
            int newX = w.getMouseX();
            int newY = w.getMouseY();
            System.out.println("Avstånd i x-led: " + (newX - oldX) + ", "
                    + "i y-led: " + (newY - oldY));
            oldX = newX;
            oldY = newY;
        }
    }
}
```

Gå igenom ett exempel ”för hand” och övertyga dig om att du förstår hur programmet fungerar.

I `SimpleWindow` kan man hantera inte bara musklick utan också tangenttryckningar. Då väntar man på en ”allmän” händelse (musklick eller tangenttryckning). När en händelse inträffar kan man testa vad som inträffade, enligt följande exempel:

```
w.waitForEvent();
if (w.getEventType() == SimpleWindow.MOUSE_EVENT) {
    // musklick, koordinaterna hämtas "som vanligt"
    int x = w.getMouseX();
    int y = w.getMouseY();
    ...
} else {
    // tangenttryckning, tecknet på tangenten hämtas med getKey
    char ch = w.getKey();
    ...
}
```

Typen `char` används för att lagra teckenvariabler (6.6 i läroboken). Man kan använda variabeln `ch` så här:

```
if (ch == 'r') {
    ... // r-tangenten trycktes ned
}
```

Uppgifter

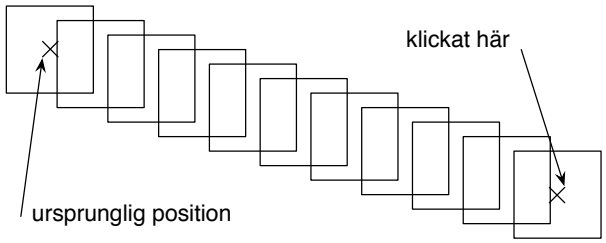
1. Kör programmen `PrintClicks` och `PrintClickDists` (i projektet *lab3*).

2. Skriv ett program där ett kvadratobjekt skapas och ritas upp i ett ritfönster. När användaren klickar med musen i fönstret ska bilden av kvadraten raderas, kvadraten flyttas till markörens position och ritas upp på nytt. Använd `waitForMouseClick()`, `getMouseX()` och `getMouseY()`.

   Välj ett lämpligt namn på klassen. För att skapa en fil för klassen kan du antingen skapa en tom klass (klicka på New Java Class-ikonen, skriv namnet på klassen) eller kopiera någon av de filer som du redan har.

   Testa programmet.

3. Modifiera programmet i uppgift 2 så att varje flyttning går till så att kvadraten flyttas stegvis till den nya positionen. Efter varje steg ska kvadraten ritas upp (utan att den gamla bilden raderas). Exempel på kvadratbilder som ritas upp när flyttningen görs i 10 steg:



Kvadratens slutliga position behöver inte bli exakt den position som man klickat på. Om man till exempel ska flytta kvadraten 94 pixlar i 10 steg är det acceptabelt att ta 10 steg med längden 9.

Skriv in och testkör programmet.

4. Ändra programmet som du skrev i uppgift 2 så att det lyssnar efter både musklick och tangenttryckningar. Vid musklick ska kvadraten flyttas precis som tidigare, om man trycker på `r`-tangenten ska kvadraterna i fortsättningen ritas med röd färg, om man trycker på någon annan tangent ska kvadraterna ritas med svart färg. Använd `SimpleWindow`-metoden `setLineColor`.

5. Någonstans i ditt program skapar du ett kvadratobjekt med en sats som har ungefär följande utseende: `Square sq = new Square(300,300,200)`. Tag bort denna sats från programmet (eller kommentera bort den). Eclipse kommer att markera flera fel i programmet, eftersom `sq` inte är deklarerad och du senare i programmet utnyttjar den variabeln. Läs och tolka felmeddelandena.

   Lägg sedan in satsen `Square sq = null` i början av programmet. Eclipse kommer inte att hitta några fel, eftersom programmet nu följer de formella reglerna för Javaprogram. Exekvera sedan programmet och se vad som inträffar. Studera felmeddelandet så att du kan tolka det.

   Meddelanden om exekveringsfel skrivs i konsolfönstret. Det skrivs också ut en ”stack trace” för felet: var felet inträffade, vilken metod som anropade metoden där det blev fel, vilken metod som anropade den metoden, osv. Om man klickar på ett radnummer öppnas rätt fil i editorn med den raden markerad (under förutsättning att programtexten, ”källkoden”, för den filen är tillgänglig).

   Lägg sedan tillbaka den ursprungliga satsen för att skapa kvadratobjektet. Ändra parametern `w` i det första anropet av `draw` till `null`. Kör programmet, studera det felmeddelande som du får.

   När man får exekveringsfel kan det vara svårt att hitta orsaken till felet. Här är en debugger till stor hjälp, i och med att man kan sätta brytpunkter och köra programmet stegvis.

6. I denna uppgift ska du lära dig fler kommandon i Eclipse. Det finns ett otal kommandon, mer eller mindre avancerade, och många av dem använder man sällan. Tre kommandon som man ofta använder:

   - Source-menyn > Correct Indentation. Korrigerar indragningarna i det markerade området i filen (gör Edit > Select All först för att markera hela filen).

- Source-menyn > Format. Korrigerar programlayouten i hela filen. Ser till exempel till att varje sats skrivs på en rad, att det är blanka runt om operatorer, och så vidare. Man bör formatera sina program regelbundet, så att de blir läsbara.
- Refactor-menyn > Rename. Ändrar namn på den variabel eller metod som markerats (lokalt om det är en lokal variabel, i hela klassen om det är ett attribut, även i andra klasser om det är en publik metod).

I laboration 2 nämnde vi att man kan ställa in hur mycket hjälp man ska få av editorn när man skriver Javaprogram. Det gör man i Preferences... i Window-menyn. Om man till exempel vill stänga av rutorna med alternativ som kommer upp under tiden man skriver så går man till Java > Editor, klickar på Content Assist och avmarkerar Enable auto activation.

Det finns hur många möjligheter som helst att ändra hur Eclipse uppför sig. Om du ändrar i Preferences och har ställt till det för dig: använd knappen Restore Defaults!

Laboration 4 — egen klass, Turtle

Mål: Du ska för första gången implementera en klass fullständigt. Du måste därför lära dig det mesta om attribut och om metoder.

Läsanvisningar

- Avsnitt 3.1–3.6 och 6.1–6.4 i läroboken.

Förberedelseuppgifter

- Gör uppgift 1.

Bakgrund

I grafiksystemet Turtle graphics kan man rita linjer. Linjerna ritas av en (tänkt) sköldpadda som går omkring i ett ritfönster. Sköldpaddan har en penna som antingen kan vara lyft (då ritas ingen linje då sköldpaddan går) eller sänkt (då ritas en linje). Sköldpaddan kan bara gå rakt framåt, i den riktning som huvudet pekar. När sköldpaddan står stilla kan den vrida sig så att dess huvud pekar åt olika håll.

En klass `Turtle` som beskriver en sköldpadda av detta slag har följande specifikation:

```
/** Skapar en sköldpadda som ritar i ritfönstret w. Från början
    befinner sig sköldpaddan i punkten x,y med pennan lyft och
    huvudet pekande rakt uppåt i fönstret (i negativ y-riktning) */
Turtle(SimpleWindow w, int x, int y);

/** Sänker pennan */
void penDown();

/** Lyfter pennan */
void penUp();

/** Går rakt framåt n pixlar i den riktning som huvudet pekar */
void forward(int n);

/** Vrider beta grader åt vänster runt pennan */
void left(int beta);

/** Går till punkten newX,newY utan att rita. Pennans läge (sänkt
    eller lyft) och huvudets riktning påverkas inte */
void jumpTo(int newX, int newY);

/** Återställer huvudriktningen till den ursprungliga */
void turnNorth();

/** Tar reda på x-koordinaten för sköldpaddans aktuella position */
int getX();

/** Tar reda på y-koordinaten för sköldpaddans aktuella position */
int getY();

/** Tar reda på sköldpaddans riktning, i grader från positiv x-led */
int getDirection();
```

Observera att vi här bestämmer vilket fönster som sköldpaddan ska rita i när vi skapar ett sköldpaddsobjekt. Det kan väl sägas motsvara verkligheten: en sköldpadda ”befinner” sig ju alltid någonstans.

I följande program ritar en sköldpadda en kvadrat med sidorna parallella med fönstrets sidor:

```
import se.lth.cs.window.SimpleWindow;

public class TurtleDrawSquare {
    public static void main(String[] args) {
        SimpleWindow w = new SimpleWindow(600, 600, "TurtleDrawSquare");
        Turtle turtle = new Turtle(w, 300, 300);
        turtle.penDown();
        for (int i = 0; i < 4; i++) {
            turtle.forward(100);
            turtle.left(90);
        }
    }
}
```

I nedanstående variant av programmet har vi gjort två ändringar: 1) längden på sköldpaddans steg väljs slumpmässigt mellan 0 och 99 pixlar, 2) efter varje steg görs en paus på 100 millisekunder. Den första ändringen medför att figuren som ritas inte blir en kvadrat, den andra ändringen medför att man ser hur varje linje ritas.

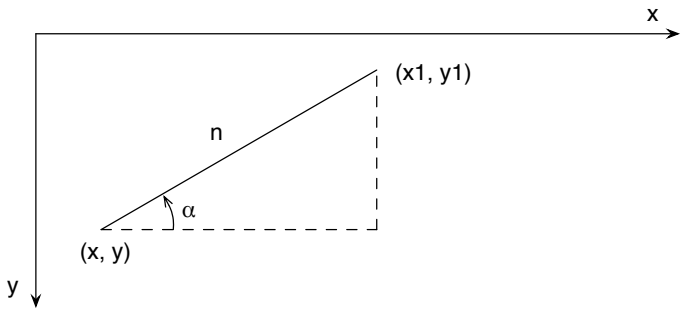
```
import se.lth.cs.window.SimpleWindow;
import java.util.Random;

public class TurtleDrawRandomFigure {
    public static void main(String[] args) {
        Random rand = new Random();
        SimpleWindow w = new SimpleWindow(600, 600, "TurtleDrawRandomFigure");
        Turtle turtle = new Turtle(w, 300, 300);
        turtle.penDown();
        for (int i = 0; i < 4; i++) {
            turtle.forward(rand.nextInt(100));
            SimpleWindow.delay(100);
            turtle.left(90);
        }
    }
}
```

Användning av slumptal beskrivs i läroboken, avsnitt 6.10. För tillfället behöver du bara veta att man måste skapa ett `Random`-objekt och att man får ett nytt slumpmässigt heltal i intervallet [0, *n*) med funktionen `nextInt(n)`. Skrivsättet [0, *n*) betyder att 0 ingår i intervallet, *n* ingår inte i intervallet. `rand.nextInt(100)` ger alltså ett slumpmässigt heltal mellan 0 och 99.

När klassen `Turtle` ska implementeras måste man bestämma vilka attribut som klassen ska ha. En sköldpadda måste hålla reda på:

- fönstret som den ska rita i: ett attribut `SimpleWindow w`,
- var i fönstret den befinner sig: en x-koordinat och en y-koordinat. För att minska inverkan av avrundningsfel i beräkningarna ska `x` och `y` vara av typ `double`,
- i vilken riktning huvudet pekar. Riktningen kommer alltid att ändras i hela grader. Man kan själv välja om attributet som anger riktningen ska vara i grader eller i radianer,
- om pennan är lyft eller sänkt. Ett sådant attribut bör ha typen `boolean`.

När sköldpaddan ska gå rakt fram i den aktuella riktningen ska en linje ritas om pennan är sänkt. Den aktuella positionen ska också uppdateras. Antag att sköldpaddan i ett visst ögonblick är vriden vinkeln α i förhållande till positiv x-led. När operationen `forward(n)` utförs ska pennan flyttas från punkten (x, y) till en ny position, som vi kallar $(x1, y1)$:



Av figuren framgår att $(x1, y1)$ ska beräknas enligt:

$$\begin{aligned} x1 &= x + n\cos\alpha \\ y1 &= y - n\sin\alpha \end{aligned}$$

I Java utnyttjas standardfunktionerna `Math.cos(alpha)` och `Math.sin(alpha)` för att beräkna cosinus och sinus. Vinkeln `alpha` ska ges i radianer.

`x` och `y` är av typ `double`. När koordinaterna utnyttjas som parametrar till `SimpleWindow`-metoderna `moveTo` och `lineTo` och när de ska returneras som funktionsresultat i `getX` och `getY` måste de avrundas till heltalsvärden. Det kan till exempel se ut så här:

```
w.moveTo((int) Math.round(x), (int) Math.round(y));
```

Uppgifter

1. I filen *Turtle.java* i projektet *lab4* finns ett ”skelett” (en klass utan attribut och med tomma metoder) till klassen `Turtle`.

   Skriv in attributen i klassen. Skriv gärna en kommentar till varje attribut som förklarar vad attributet betyder. Skriv också konstruktorn och se till att alla attribut får rätt startvärden. Implementera operationerna `penDown`, `forward` och `left`.

2. Klasserna `TurtleDrawSquare` och `TurtleDrawRandomFigure` finns i filerna *TurtleDrawSquare.java* och *TurtleDrawRandomFigure.java*. Kör programmen och kontrollera att din `Turtle`-implementering är korrekt.

3. Implementera de återstående metoderna i klassen `Turtle`. Kör programmet `TestTurtle` — det testar alla metoderna utom `penDown` och `penUp`.

4. Om man har flera sköldpaddor som ritar samtidigt så måste man tänka på att sköldpaddorna ritar med samma ”penna”. Kör programmet `TestFourTurtles` — i fönstret ska det ritas fyra kvadrater; om resultatet blir något annat är det något fel i din `Turtle`-klass.

Laboration 5 — mera Turtle, arv

Mål: Du ska lära dig att skapa projekt i Eclipse och skriva program som använder din `Turtle`-klass från laboration 4. Du ska också få en introduktion till strukturering av klasser med hjälp av arv (kapitel 9 i läroboken) — detta går vi igenom ordentligt senare i kursen.

Läsanvisningar

- Avsnitt 6.10 (slumptal) i läroboken. Titta också gärna på avsnitt 9.1, om arv.
- Beskrivningen av klassen `Color` i appendix C i läroboken.

Förberedelseuppgifter

- Gör uppgift 1–4.

Bakgrund

I en av uppgifterna i laborationen ska du använda en klass `ColorTurtle` som beskriver en sköldpadda som ritar med en färgad penna. För övrigt fungerar klassen precis likadant som en ”vanlig” `Turtle`. Man skapar objekt av klassen enligt följande exempel:

```
ColorTurtle t1 = new ColorTurtle(w, 100, 100, java.awt.Color.RED);
ColorTurtle t2 = new ColorTurtle(w, 100, 100, java.awt.Color.BLUE);
```

Det är inte särskilt svårt att skriva klassen: den enda skillnaden mot `Turtle` är att klassen behöver ett attribut där färgen sparas och att man i metoden `forward` sätter linjefärgen till denna färg med `SimpleWindow`-operationen `setLineColor`. Man kan kopiera klassen `Turtle` till `ColorTurtle` och införa dessa tillägg. Detta är dock inte alls bra — om man senare kommer på att man måste ändra något i `Turtle` måste man komma ihåg att också ändra samma sak i `ColorTurtle`.

Betydligt bättre är att använda *arv:* man låter `ColorTurtle` ”ärva” alla egenskaper från `Turtle` och skriver bara de tillägg som behövs. Man kallar `ColorTurtle` för en ”subklass” till ”superklassen” `Turtle`.

Vi kommer att gå igenom arv noggrant senare i kursen. Här visar vi bara hur man skriver `ColorTurtle`:

```
import se.lth.cs.window.SimpleWindow;
import java.awt.Color;
import lab4.Turtle;

public class ColorTurtle extends Turtle {
    private Color color;

    public ColorTurtle(SimpleWindow w, int x, int y, Color color) {
        super(w, x, y);
        this.color = color;
    }

    public void forward(int n) {
        Color savedColor = w.getLineColor();
        w.setLineColor(color);
        super.forward(n);
        w.setLineColor(savedColor);
    }
}
```

Kommentarer:

- `extends`, ”utökar”, anger att klassen `ColorTurtle` ärver alla egenskaper från klassen `Turtle`.
- Alla objektets attribut måste få startvärden. De attribut som finns i ”`Turtle`-delen” av objektet initierar man genom att anropa `Turtle`-konstruktorn med `super(w, x, y)`.
- I `forward` sparar man den aktuella linjefärgen, ändrar till sköldpaddans egen färg, anropar `forward` i superklassen `Turtle` och sätter tillbaka linjefärgen.
- För att det ska fungera måste man göra en ändring i `Turtle`: skyddet på `SimpleWindow`-attributet `w` ändras från `private` till `protected`. Det betyder att subklasserna får använda attributet.

Uppgifter

1. Till denna uppgift finns inga färdigskrivna klasser och därför inte något Eclipseprojekt. Du måste alltså skapa ett sådant:

   1. Välj File > New > Java Project.
   2. Skriv namnet på projektet (*lab5*).
   3. Klicka på Finish.

2. Klasserna som du skriver kommer att använda klassen `SimpleWindow`. Denna klass finns i biblioteksfilen *cs_eda016.jar*, som finns i projektet *cs_eda016*.

   Du måste ange att projektet utnyttjar denna fil:

   1. Högerklicka på projektet *lab5*, välj Build Path > Configure Build Path.
   2. Klicka på Libraries.
   3. Klicka på Add JARS. . . , öppna projektet *cs_eda016*, välj filen *cs_eda016.jar*.

3. Dina program kommer att använda klassen `Turtle` från laboration 4. Du måste ange att filerna i projektet *lab4* ska vara tillgängliga i detta projekt:

   1. Högerklicka på projektet *lab5*, välj Build Path > Configure Build Path.
   2. Klicka på Projects.
   3. Klicka på Add. . . , välj projektet *lab4*.

4. Skriv ett program där en sköldpadda tar 1000 steg i ett fönster. Sköldpaddan ska börja sin vandring mitt i fönstret. I varje steg ska steglängden väljas slumpmässigt i intervallet $[1, 10]$. Efter varje steg ska sköldpaddan vridas ett slumpmässigt antal grader i intervallet $[-180, 180)$.

5. Skriv ett program där *två* sköldpaddor vandrar över ritfönstret. Steglängden och vridningsvinkeln ska väljas slumpmässigt i samma intervall som i föregående uppgift. Vandringen ska avslutas när avståndet mellan de båda sköldpaddorna är mindre än 50 pixlar. Sköldpaddorna ska turas om att ta steg på följande sätt:

   ```
   while ("avståndet mellan sköldpaddorna" >= 50) {
       "låt den ena sköldpaddan ta ett slumpmässigt steg och göra
        en slumpmässig vridning"
       "låt den andra sköldpaddan ta ett slumpmässigt steg och göra
        en slumpmässig vridning"
       SimpleWindow.delay(10);
   }
   ```

   Låt den ena sköldpaddan börja sin vandring i punkten (250,250) och den andra i (350,350).

6. På kurshemsidan, under Datorlaborationer, finns en länk till filen *ColorTurtle.java* som innehåller klassen `ColorTurtle`. Ladda ner filen, importera den till projektet *lab5* (Import > General > File System). Ändra i din `Turtle`-klass i projektet *lab4* så att `SimpleWindow`-attributet får rätt skydd. Ändra sedan slumpvandringsprogrammet från uppgift 5 så att en röd och en blå sköldpadda vandrar i fönstret.

Laboration 6 — labyrint

Mål: Du lösa ett problem där huvuduppgiften är att konstruera en algoritm för att hitta vägen genom en labyrint. Du ska använda arv även i denna uppgift.

Läsanvisningar

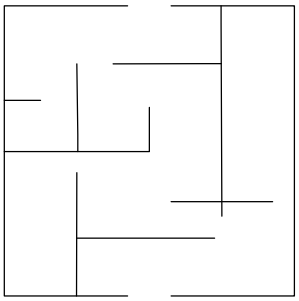
- —

Förberedelseuppgifter

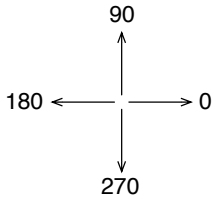
- Gör uppgift 1 och 3, skissa på algoritmen i uppgift 2.

Bakgrund

En labyrint består av ett rum med en ingång och en utgång. Alla väggar är parallella med x- eller y-axeln. Ingången finns alltid i den vägg som är längst ner i figuren. Exempel:



Ett sätt att hitta från ingången till utgången är att gå genom labyrinten och hela tiden hålla den vänstra handen i väggen. När man vandrar genom labyrinten är fyra riktningar möjliga. En riktning definieras som vinkeln mellan x-axeln och vandringsriktningen och mäts i grader:



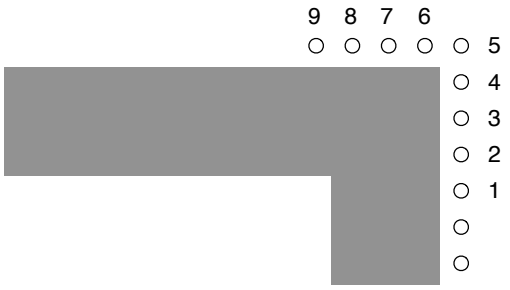
Labyrinten beskrivs av en färdigskriven klass `Maze`. Labyrintens utseende läses in från en fil när labyrintobjektet skapas. Klassen har följande specifikation:

```
/** Skapar en labyrint med nummer nbr */
Maze(int nbr);

/** Ritar labyrinten i fönstret w */
void draw(SimpleWindow w);

/** Tar reda på x-koordinaten för ingången */
int getXEntry();

/** Tar reda på y-koordinaten för ingången */
int getYEntry();
```

```
/** Undersöker om punkten x,y är vid utgången */
boolean atExit(int x, int y);

/** Undersöker om man, när man befinner sig i punkten x,y och är på väg i
    riktningen direction, har en vägg direkt till vänster om sig */
boolean wallAtLeft(int direction, int x, int y);

/** Som wallAtLeft, men undersöker om man har en vägg direkt framför sig */
boolean wallInFront(int direction, int x, int y);
```

I operationerna `wallAtLeft` och `wallInFront` betraktas alla riktningar $R \pm n \cdot 360°, n = 1, 2, \ldots$ som ekvivalenta med R.

Väggarna har bredden 6 pixlar. De ritas dock med bredden 4 pixlar för att det ska bli litet luft mellan sköldpaddan och väggen. Exempel där man vandrar runt ett hörn:



I punkterna 1–4 är riktningen 90. `wallAtLeft` ger i dessa punkter värdet `true`, `wallInFront` värdet `false`. I punkt 5 ger `wallAtLeft` värdet `false`, varför man svänger åt vänster (riktningen 90 ändras till 180). Man fortsätter sedan genom punkterna 6–9. I dessa punkter ger `wallAtLeft` värdet `true`, `wallInFront` värdet `false`.

I uppgiften ska du skriva en klass `MazeTurtle` som fungerar precis som en ”vanlig” `Turtle` med tillägget att den också kan vandra i en labyrint. Vandringen ska göras i en metod med följande rubrik:

```
/** Låter sköldpaddan vandra genom labyrinten maze, från
    ingången till utgången */
public void walk(Maze maze);
```

Använd arv för att lägga till metoden `walk`. I metoden har du då direkt tillgång till alla metoder som ärvts från `Turtle`: `forward`, `left`, och så vidare.

Uppgifter

1. Skapa ett projekt *lab6* (se uppgift 1, laboration 5). Gå in i Build Path och markera att *cs_eda016.jar* och projektet *lab4* används (uppgift 2–3, laboration 5).

2. Skriv klassen `MazeTurtle`. I metoden `walk` ska sköldpaddan börja sin vandring i punkten med koordinaterna `getXEntry()`, `getYEntry()` och gå uppåt i labyrinten. Alla steg ska ha längden 1. Lägg in en fördröjning efter varje steg så att man ser hur sköldpaddan vandrar. Klassen `Maze` finns i paketet `se.lth.cs.ptdc.maze`.

3. Skriv ett huvudprogram som kontrollerar att sköldpaddan kan vandra i labyrinterna. Numret på labyrinten ska läsas in från tangentbordet.

   Testa programmet. Det finns fem olika labyrinter, med nummer 1–5. Din sköldpadda bör klara av alla labyrinterna, men du blir godkänd även om den inte klarar labyrint nummer 5.

Laboration 7 — simulering av patiens

Mål: Du ska skriva ett enkelt simuleringsprogram och lära dig att använda vektorer. Du ska också träna på att använda Eclipse debugger.

Läsanvisningar

- Avsnitt 8.1–8.3 i läroboken.
- Avsnittet om Eclipse debugger i bilaga C.

Förberedelseuppgifter

- Gör uppgift 1 och 2.

Bakgrund

Patiensen 1–2–3 läggs på följande sätt: Man tar en kortlek, blandar den och lägger sedan ut korten ett efter ett. Samtidigt som man lägger korten räknar man 1–2–3–1–2–..., det vill säga när man lägger det första kortet säger man 1, när man lägger det andra kortet säger man 2, osv. Patiensen går ut om man lyckas lägga ut alla kort i leken utan att någon gång få upp ett ess när man säger 1, någon 2-a när man säger 2 eller någon 3-a när man säger 3.

Man kan med hjälp av sannolikhetslära bestämma exakt hur stor sannolikheten är att patiensen ska gå ut.[1] Men det är betydligt enklare att uppskatta sannolikheten genom att lägga patiensen många gånger och räkna antalet gånger som den går ut.

Allra snabbast går det om datorn lägger patiensen. Då har man användning av följande klasser, som beskriver kort och kortlekar:

```
/** konstanter för färgerna */
static final int SPADES = ...;
static final int HEARTS = SPADES + 1;
static final int DIAMONDS = SPADES + 2;
static final int CLUBS = SPADES + 3;

/** Skapar ett spelkort med färgen suit (SPADES, HEARTS, DIAMONDS, CLUBS)
    och valören rank (1-13) */
Card(int suit, int rank);

/** Tar reda på färgen */
int getSuit();

/** Tar reda på valören */
int getRank();
```

```
/** Skapar en kortlek */
CardDeck();

/** Blandar kortleken */
void shuffle();

/** Undersöker om det finns fler kort i kortleken */
boolean moreCards();

/** Drar det översta kortet i leken */
Card getCard();
```

[1] Se ”Fråga Lund om matematik”, `www.maths.lth.se/query/answers/`, sök efter ”123-patiens” bland frågorna.

I följande program skriver man ut färg och valör för alla kort som man drar ur en blandad kortlek:

```
import se.lth.cs.ptdc.cardGames.Card;
import se.lth.cs.ptdc.cardGames.CardDeck;

public class CardExample {
    public static void main(String[] args) {
        CardDeck deck = new CardDeck();
        deck.shuffle();
        while (deck.moreCards()) {
            Card c = deck.getCard();
            String suitString = "";
            switch (c.getSuit()) {
                case Card.SPADES:   suitString = "Spades"; break;
                case Card.HEARTS:   suitString = "Hearts"; break;
                case Card.DIAMONDS: suitString = "Diamonds"; break;
                case Card.CLUBS:    suitString = "Clubs"; break;
            }
            System.out.println(c.getRank() + " of " + suitString);
        }
    }
}
```

Uppgifter

1. Skapa ett projekt *lab7* (se uppgift 1, laboration 5). Gå in i Build Path och markera att *.jar*-filen *cs_eda016.jar* utnyttjas (uppgift 2, laboration 5).

2. Skriv ett program som lägger patiensen 1–2–3 ett stort antal gånger och skriver ut sannolikheten för att patiensen ska gå ut. Den korrekta sannolikheten är ungefär 0.008165. Använd de färdigskrivna klasserna `Card` och `CardDeck`.

3. Implementera klassen `CardDeck`. Tag ur huvudprogrammet bort satsen där den färdigskrivna klassen `CardDeck` importeras. Kör programmet på nytt och kontrollera att du får ungefär samma resultat som i uppgift 2.

   I klassen ska man hålla reda på de 52 korten i kortleken. Det gör man enklast genom att lagra korten i en vektor:

   ```
   import se.lth.cs.ptdc.cardGames.Card;
   import java.util.Random;

   class CardDeck {
       private Card[] cards; // korten
       private int current;  // index för "nästa" kort
       private static Random rand = new Random();

       /** Skapar en kortlek */
       public CardDeck() {
           cards = new Card[52]; // skapa vektorn
           ...                   // skapa korten, lägg in dem i vektorn
           current = 0;
       }

       ...
   }
   ```

Att skapa korten i ordning är inte så enkelt — man känner ju inte värdet på konstanterna som anger färgerna. Men man ser att konstanterna ligger i ordning, så man kan skriva satser som den följande:

```
for (int suit = Card.SPADES; suit <= Card.CLUBS; suit++) { ... }
```

För att blanda kortleken bör man använda följande algoritm:

```
för i = 51, 50, ..., 1 {
    nbr = slumptal i intervallet [0,i+1)
    byt plats på korten med index i och index nbr
}
```

Laboration 8 — bildbehandling, del 1

Mål: Du ska lära dig att använda matriser och att implementera mera komplicerade operationer. Du ska också få en introduktion till bildbehandling.

Läsanvisningar

- Avsnitt 8.6 (matriser), 9.3 och 9.6 (arv) i läroboken.

Förberedelseuppgifter

- Gör uppgift 1–3.

Bakgrund

En digital bild består av ett rutnät (en matris) av pixlar. Varje pixel har en färg, och om man har många pixlar flyter de samman för ögat så att de tillsammans skapar en bild.

Färger kan anges enligt olika system, men i datorer är det vanligt att man använder RGB-systemet. I RGB-systemet sätts en färg samman av tre grundfärger: röd, grön och blå. Mättnaden av varje grundfärg anges av ett heltal som vi i fortsättningen förutsätter ligger i intervallet $[0, 255]$. 0 anger ”ingen färg” och 255 anger ”maximal färg”. Man kan därmed representera $256 \times 256 \times 256 = 16\,777\,216$ olika färgnyanser. Man kan också representera gråskalor; det gör man med färger som har samma värde på alla tre grundfärgerna: $(0, 0, 0)$ är helt svart, $(255, 255, 255)$ är helt vitt.

I Java representeras färger av klassen `java.awt.Color`. Den klassen beskrivs i appendix C i läroboken och utförligare i Javadokumentationen på nätet.

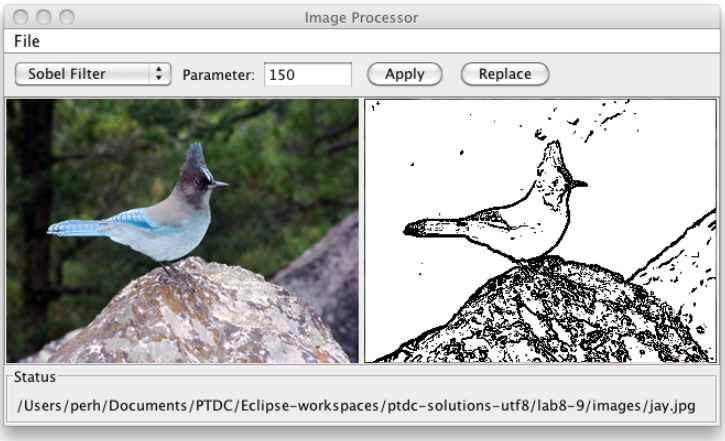
När man har en digital bild i datorn kan man förändra och förbättra bilden på olika sätt. Det finns många program för detta ändamål, till exempel Photoshop (kommersiellt) och GIMP (gratis, GNU-projekt). Du ska i denna uppgift skriva ett program som kan göra en del enkla transformationer (kallas i fortsättningen ”filtreringar”) av bilder, till exempel förbättra kontrasten i en bild och markera konturer i en bild. Användargränssnittet till programmet är färdigskrivet; din uppgift är att skriva metoder som utför filtreringarna.

Figur 1 på följande sida visar hur användargränssnittet till programmet ser ut. Man väljer filtreringsmetod i menyn längst till vänster och utför filtreringen genom att klicka på Apply-knappen. En del av filtreringsmetoderna behöver en parameter; värdet på denna skriver man i textrutan. I fönstret visas originalbilden till vänster och den filtrerade bilden till höger (här har Sobelfiltrering använts, som markerar konturer i bilden).

Användargränssnittet beskrivs av klassen `ImageGUI`. `main`-metoden finns i klassen `ImageProcessor`. Den klassen har följande utseende:

```
import se.lth.cs.ptdc.images.ImageFilter;
import se.lth.cs.ptdc.images.ImageGUI;

public class ImageProcessor {
    public static void main(String[] args) {
        ImageFilter[] filters = new ImageFilter[1];
        filters[0] = new IdentityFilter("Identity Filter");
        new ImageGUI(filters);
    }
}
```

De objekt som man lägger in i vektorn `filters` tas omhand av användargränssnittet. Du kommer senare att skapa flera filterobjekt av olika slag och skicka till `ImageGUI`. Namnen på filtren (strängparametern till konstruktorn) visas i menyn där man väljer filtreringsmetod. När man trycker



Figur 1: Användargränssnittet i bildbehandlingsprogrammet.

på Apply-knappen letar `ImageGUI` upp filtreringsobjektet med rätt namn och anropar en metod med namnet `apply` i detta objekt. Om man trycker på Replace-knappen ersätts originalbilden av den filtrerade bilden (det använder man om man vill filtrera en bild med flera olika filter efter varandra).

Det finns en färdigskriven filterklass, `IdentityFilter`. Filtreringsmetoden i den klassen gör ingenting annat än att skapa en identisk kopia av ursprungsbilden. Klassen finns med bara för att visa hur en filterklass är uppbyggd:

```
import java.awt.Color;
import se.lth.cs.ptdc.images.ImageFilter;

/** IdentityFilter beskriver en identitetstransformation */
public class IdentityFilter extends ImageFilter {
    /** Skapar ett filterobjekt med namnet name */
    public IdentityFilter(String name) {
        super(name);
    }

    /** Filtrerar bilden i matrisen inPixels och returnerar resultatet
        i en ny matris. Utnyttjar eventuellt värdet av paramValue */
    public Color[][] apply(Color[][] inPixels, double paramValue) {
        int height = inPixels.length;
        int width = inPixels[0].length;
        Color[][] outPixels = new Color[height][width];
        for (int i = 0; i < height; i++) {
            for (int j = 0; j < width; j++) {
                Color pixel = inPixels[i][j];
                outPixels[i][j] = new Color(pixel.getRed(),
                                            pixel.getGreen(),
                                            pixel.getBlue());
            }
        }
        return outPixels;
    }
}
```

Metoden `apply` har två parametrar: `inPixels` som är originalbilden (matrisen med `Color`-objekt) och `paramValue` som är det värde som man skrivit i Parameter-rutan. (Om man inte har skrivit något i Parameter-rutan, eller skrivit något som inte kan tolkas som ett talvärde, så har `paramValue` värdet 0.) `paramValue`-värdet används inte i klassen `IdentityFilter`.

I metoden skapas en matris `outPixels` med samma storlek som `inPixels`. Därefter skapas ett nytt `Color`-objekt i varje matriselement. Eftersom det är en identitetstransformation så är varje pixel i `outPixels` lika med motsvarande pixel i `inPixels` (de har samma RGB-värden). Så kommer det naturligtvis inte att vara i de filtreringsmetoder som du ska skriva: där kommer RGB-värdena i `outPixels` att skilja sig från motsvarande värden i `inPixels`. Till slut returneras den nya matrisen. `ImageGUI` omvandlar sedan matrisen till en bild och visar den i användargränssnittet.

I superklassen `ImageFilter` beskrivs det som är gemensamt för alla filter: att de har ett namn och en metod `apply` som behandlar en bild. Dessutom finns två hjälpmetoder i klassen. Att metoden `apply` är ”abstrakt” innebär att den måste implementeras i alla subklasser. Klassen har följande utseende:

```
import java.awt.Color;

/** ImageFilter är superklass till alla filterklasser */
public abstract class ImageFilter {
    private String name;

    /** Skapar ett filterobjekt med namnet name */
    protected ImageFilter(String name) {
        this.name = name;
    }

    /** Tar reda på filtrets namn */
    public String getName() {
        return name;
    }

    /** Filtrerar bilden i matrisen inPixels och returnerar resultatet
        i en ny matris. Utnyttjar eventuellt värdet av paramValue.
        Abstrakt metod, implementeras i subklasserna */
    public abstract Color[][] apply(Color[][] inPixels, double paramValue);

    /** Beräknar intensiteten hos alla pixlarna i pixels, returnerar
        resultatet i en ny matris */
    protected short[][] computeIntensity(Color[][] pixels) {
        // ... färdigskriven metod, se uppgift 5 nedan
    }

    /** Faltar punkten p[i][j] med faltningskärnan kernel. Summan
        av elementen i kernel är weight */
    protected short convolve(short[][] p, int i, int j,
                             short[][] kernel, int weight) {
        // ... färdigskriven metod, se uppgift 2, laboration 9
    }
}
```

Uppgifter

1. Under Dokumentation på kurshemsidan finns en länk till API-specifikationerna för alla standardklasserna i Java. Leta upp specifikationen av klassen `Color` i paketet `java.awt` och titta igenom den.

2. I projektet *lab8-9* finns filerna *ImageProcessor.java* och *IdentityFilter.java*. Det finns också en katalog *images* som innehåller ett antal JPEG-bilder som du kan testa programmet med (du får naturligtvis använda egna JPEG-bilder om du vill).

Testkör programmet. I File-menyn finns kommandon för att öppna bildfiler och för att spara filtrerade bilder på fil. Använd nya filnamn om du sparar filtrerade bilder.

3. **Blåfilter.** Skriv en klass `BlueFilter` som skapar en ny bild där varje pixel bara innehåller den blå komponenten. Kopiera klassen `IdentityFilter` och gör de ändringar som behövs. Ändra också huvudprogrammet så att det nya filtret kommer med i menyn.

4. **Invertering av bild.** Skriv en klass `InvertFilter` som inverterar en bild dvs skapar en ”negativ” kopia av bilden.

   Fundera över vad som kan menas med en inverterad eller negativ kopia: de nya RGB-värdena är inte ett dividerat med de gamla värdena (då skulle de nya värdena bli flyttal) och inte de gamla värdena med ombytt tecken (då skulle de nya värdena bli negativa).

5. **Konvertering av färgbild till gråskalebild.** Skriv en klass `GrayScaleFilter` som konverterar en färgbild till en gråskalebild. (Om man utgår från en gråskalebild ska resultatet bli identiskt med ursprungsbilden.)

   Här gäller det alltså att omvandla varje punkt i RGB-rymden (som har storleken $256 \times 256 \times 256$) till en punkt i gråskalerymden (som har storleken 256). Man behöver ett mått på hur ”ljus” eller ”intensiv” varje färg är. Om vi kallar RGB-komponenterna r, g respektive b får vi ett mått på intensiteten genom att räkna ut avståndet från punkten (r, g, b) till origo.

   Man kan använda den vanliga avståndsformeln:

   ```
   intensity = Math.sqrt((r * r + g * g + b * b) / 3)
   ```

   Divisionen med 3 kommer sig av att man vill att resultatet ska hamna i intervallet $[0, 255]$. Det blir dock tidsödande att göra tre multiplikationer och en kvadratrotsberäkning för varje pixel, så man kan i stället använda följande mått:

   ```
   intensity = (r + g + b) / 3
   ```

   Intensiteter måste beräknas i flera av uppgifterna och man bör beskriva hur man gör det bara en gång. Den färdigskrivna metoden `short[][] computeIntensity(Color[][])` som utför beräkningarna har därför placerats i klassen `ImageFilter`. Den blir därigenom tillgänglig i alla subklasser. Skyddsspecifikationen `protected` betyder att metoden är tillgänglig i subklasserna men inte utanför klassen.

   Att göra en gråskalefiltrering tar inte irriterande lång tid, men vi ska ändå titta på en uppenbar optimering som sparar både tid och minnesutrymme. Gör denna optimering om du vill och undersök om filtreringen blir märkbart snabbare.

   En gråskalebild innehåller högst 256 olika färger: $(0, 0, 0), (1, 1, 1), \ldots, (255, 255, 255)$. Om vi antar att en viss bild har 300×500 pixlar och att man skapar ett nytt `Color`-objekt för varje pixel så blir det totalt 150 000 objekt, varav många är identiska. Man kan klara sig med 256 olika objekt om man börjar med att skapa en vektor med de möjliga färgerna:

   ```
   Color[] grayLevels = new Color[256];
   // skapa färgen (0,0,0) och lägg in den i grayLevels[0],
   // (1,1,1) i grayLevels[1], ..., (255,255,255) i grayLevels[255]
   ```

   När man sedan har bestämt sig för att en pixel ska ha en viss intensitet 0–255 behöver man inte skapa ett nytt objekt med `new Color(intensity, intensity, intensity)` — ett sådant objekt finns ju färdigt i vektorelementet `grayLevels[intensity]`.

Laboration 9 — bildbehandling, del 2

Mål: Du ska skriva mer avancerade filtreringsalgoritmer till bildbehandlingsprogrammet `ImageProcessor`. För att bli godkänd på laborationen måste du klara av uppgift 1. De båda andra uppgifterna är extrauppgifter som du bör lösa om du har tid över.

Läsanvisningar

- Avsnitt 8.6 (matriser) och 8.10 (registrering) i läroboken.

Förberedelseuppgifter

- Gör uppgift 1, till och med beräkningen av intensitetshistogrammet.

Uppgifter

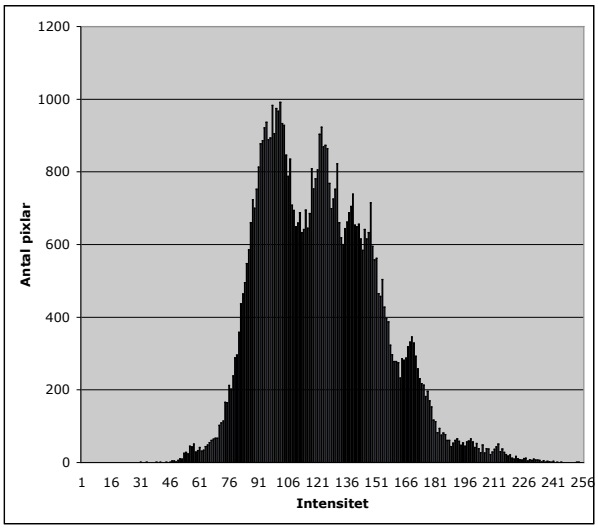
1. **Förbättring av kontrasten i bild.** Vi inskränker oss här till att förbättra kontrasten i gråskalebilder. Om man applicerar kontrastfiltrering på en färgbild så kommer bilden att konverteras till en gråskalebild. (Man kan naturligtvis förbättra kontrasten i en färgbild och få en färgbild som resultat. Då behandlar man de tre färgkanalerna var för sig.)

   Många bilder lider av alltför låg kontrast. Det beror på att bilden inte utnyttjar hela det tillgängliga området 0–255 för intensiteten. I diagrammet i figur 2 visas antalet pixlar med olika intensiteter för bilden *images/girl2.jpg*. Ett sådant diagram kallas ett *intensitetshistogram*.

   Här ser vi att det inte finns några pixlar med intensitet mindre än 45 och bara få pixlar med intensitet större än 225 (ungefär). Man får en bild med bättre kontrast om man ”töjer ut” intervallet enligt följande formel (lineär interpolation):

   ```
   newIntensity = 255 * (intensity - 45) / (225 - 45)
   ```

   Som synes kommer en punkt med intensiteten 45 att få den nya intensiteten 0 och en punkt med intensiteten 225 att få den nya intensiteten 255. Mellanliggande punkter sprids ut jämnt över intervallet $[0, 255]$. För punkter med en intensitet mindre än 45 sätter man den nya intensiteten till 0, för punkter med en intensitet större än 225 sätter man den nya intensiteten till 255.



Figur 2: Intensitetshistogram.

Vi kallar intervallet där de flesta pixlarna finns för `[lowCut, highCut]`. De punkter som har intensitet mindre än `lowCut` sätter man till 0, de som har intensitet större än `highCut` sätter man till 255. För de övriga punkterna interpolerar man med formeln ovan (45 ersätts med `lowCut`, 225 med `highCut`).

Det återstår nu att hitta lämpliga värden på `lowCut` och `highCut`. Detta är inte något som kan göras helt automatiskt, eftersom värdena beror på intensitetsfördelningen hos bildpunkterna. Man börjar med att beräkna bildens intensitetshistogram, dvs hur många punkter i bilden som har intensiteten 0, hur många som har intensiteten 1, ..., till och med 255. Detta är ett typiskt registreringsproblem som ska lösas enligt metoden i avsnitt 8.10 i läroboken.

I de flesta bildbehandlingsprogram kan man sedan titta på histogrammet och interaktivt bestämma värdena på `lowCut` och `highCut`. Så ska vi dock inte göra här. I stället bestämmer vi oss för ett procenttal `cutOff` (som vi matar in i Parameter-rutan i användargränssnittet) och beräknar `lowCut` så att `cutOff` procent av punkterna i bilden har en intensitet som är mindre än `lowCut` och `highCut` så att `cutOff` procent av punkterna har en intensitet som är större än `highCut`.

Exempel: antag att en bild innehåller 100 000 pixlar och att `cutOff` är 1.5. Beräkna bildens intensitetshistogram i en vektor `int[] histogram = new int[256]`. Beräkna `lowCut` så att `histogram[0] + histogram[1] + ... + histogram[lowCut] = 0.015 * 100000` (så nära det går att komma, det blir troligen inte exakt likhet). Beräkna `highCut` på liknande sätt.

Sammanfattning av algoritmen:

1. Beräkna intensiteterna hos alla punkterna i bilden, lagra dem i en `short`-matris. Använd den färdigskrivna metoden `computeIntensity`.
2. Beräkna bildens intensitetshistogram.
3. Parametervärdet `paramValue` är det värde som ska användas som `cutOff`.
4. Beräkna `lowCut` och `highCut` enligt ovan.
5. Beräkna nya intensiteter enligt interpolationsformeln och lagra de nya pixlarna i `outPixels`.

Skriv en klass `ContrastFilter` som implementerar algoritmen. I katalogen *images* kan bilderna *moon.jpg*, *girl1.jpg* och *girl2.jpg* vara lämpliga att testa, eftersom de är bilder med låg kontrast.

Anmärkning: om `cutOff` sätts $= 0$ så får man samma resultat av denna filtrering som man får av algoritmen i uppgift 5, laboration 8. Det inser man lätt genom att studera interpolationsformeln.

2. **Utjämning av konturer, Gaussfiltrering.** Gaussfiltrering är ett exempel på så kallad *faltningsfiltrering*. (Du kommer att läsa mycket om faltning i kommande kurser i matematik och signalbehandling.) Filtreringen bygger på att man modifierar varje bildpunkt genom att titta på punkten och omgivande punkter.

För detta utnyttjar man en så kallad faltningskärna K som är en liten kvadratisk heltalsmatris. Man placerar K över varje element i intensitetsmatrisen och multiplicerar varje element i K med motsvarande element i intensitetsmatrisen. Man summerar produkterna och dividerar summan med summan av elementen i K för att få det nya värdet på intensiteten i punkten. Divisionen med summan gör man för att de nya intensiteterna ska hamna i rätt intervall.

Exempel:

$$intensity = \begin{pmatrix} 5 & 4 & 2 & 8 & \dots \\ 4 & 3 & 4 & 9 & \dots \\ 9 & 8 & 7 & 7 & \dots \\ 8 & 6 & 6 & 5 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix} \qquad K = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 4 & 1 \\ 0 & 1 & 0 \end{pmatrix}$$

Här är summan av elementen i K $1+1+4+1+1=8$. För att räkna ut det nya värdet på intensiteten i punkten med index `[1][1]` (det nuvarande värdet är 3) beräknar man:

$$newintensity = \frac{0\cdot 5+1\cdot 4+0\cdot 2+1\cdot 4+4\cdot 3+1\cdot 4+0\cdot 9+1\cdot 8+0\cdot 7}{8} = \frac{32}{8} = 4$$

Man fortsätter med att flytta K ett steg åt höger och beräknar på motsvarande sätt ett nytt värde för elementet med index `[1][2]` (där det nuvarande värdet är 4 och det nya värdet blir 5). Därefter gör man på samma sätt för alla element utom för ”ramen” dvs elementen i matrisens ytterkanter.

Skriv en klass `GaussFilter` som implementerar denna algoritm. Varje färg ska behandlas separat. Gör på följande sätt:

1. Bilda tre `short`-matriser och lagra pixlarnas red-, green- och blue-komponenter i matriserna.
2. Utför faltningen av de tre komponenterna för varje element och lagra ett nytt `Color`-objekt i `outPixels` för varje punkt.
3. Elementen i ramen behandlas ju inte, men i `outPixels` måste också dessa element få värden. Enklast är att flytta över dessa element oförändrade från `inPixels` till `outPixels`. Man kan också sätta dem till `Color.WHITE`, men då kommer den filtrerade bilden att se något mindre ut.

I `ImageFilter` finns en färdigskriven metod med följande rubrik:

```
protected short convolve(short[][] p, int i, int j,
                         short[][] kernel, int weight);
```

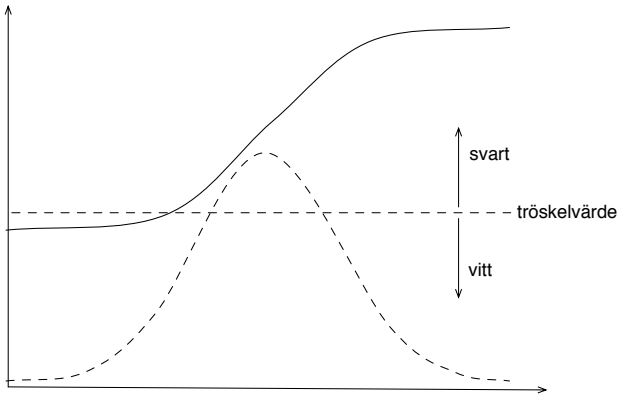
Metoden faltar punkten `p[i][j]` med faltningskärnan `kernel` och ska anropas med red-, green- och blue-matrisen. `weight` är summan av elementen i `kernel`. Faltningskärnan kan vara ett attribut i klassen och skapas enklast på följande sätt:

```
private static short[][] GAUSS_KERNEL =
    { {0, 1, 0},
      {1, 4, 1},
      {0, 1, 0} };
```

Det kan vara intressant att prova med andra värden än 4 i mitten av faltningsmatrisen. Med värdet 0 får man en större utjämning eftersom man då inte alls tar hänsyn till den aktuella pixelns värde. Mata in detta värde i Parameter-rutan.

Anmärkning: det kan ibland vara svårt att se någon skillnad mellan den filtrerade bilden och originalbilden. Om man vill ha en riktigt suddig bild så måste man använda en större matris som faltningskärna.

3. **Markering av konturer, Sobelfiltrering.** Med Gaussfiltrering beräknar man medelvärden och jämnar därför ut en bild. Med Sobelfiltrering, som också är ett exempel på faltningsfiltrering, får man motsatt effekt dvs man förstärker konturer i en bild. I princip deriverar man bilden i x- och y-led och sammanstället resultatet.



Figur 3: En funktion (heldragen linje) och dess derivata (streckad linje).

Först en förklaring om varför derivering förstärker konturer. I figur 3 visas en funktion f (heldragen linje) och funktionens derivata f' (streckad linje). Vi ser att där funktionen gör ett ”hopp” så får derivatan ett stort värde. Om funktionen representerar intensiteten hos pixlarna längs en linje i x-led eller y-led så motsvarar ”hoppen” en kontur i bilden. Om man sedan bestämmer sig för att pixlar där derivatans värde överstiger ett visst tröskelvärde ska vara svarta och andra pixlar vita så får man en bild med bara konturer.

Nu är ju intensiteten hos pixlarna inte en kontinuerlig funktion som man kan derivera enligt vanliga matematiska regler. Men man kan approximera derivatan, till exempel med följande formel:

$$f'(x) \approx \frac{f(x+h) - f(x-h)}{2h}$$

(Om man här låter h gå mot noll så får man definitionen av derivatan.) Uttryckt i Java och matrisen `intensity` så får man:

```
derivative = (intensity[i][j+1] - intensity[i][j-1]) / 2
```

Allt detta kan man uttrycka med hjälp av faltning.

1. Beräkna intensitetsmatrisen med metoden `computeIntensity`.
2. Falta varje punkt i intensitetsmatrisen med *två* kärnor:

   ```
   X_SOBEL = { {-1, 0, 1},          Y_SOBEL = { {-1, -2, -1},
               {-2, 0, 2},                      { 0,  0,  0},
               {-1, 0, 1} };                    { 1,  2,  1} };
   ```

   Använd metoden `convolve` med vikten 1 (här behöver man inte normera resultatet). Koefficienterna i matrisen `X_SOBEL` uttrycker derivering i x-led (ger vertikala konturer), i `Y_SOBEL` faltning i y-led (ger horisontella konturer). För att förklara varför koefficienterna ibland är 1 och ibland 2 måste man studera den bakomliggande teorin noggrant, men det gör vi inte här.
3. Om resultaten av faltningen i en punkt betecknas med `sx` och `sy` så får man en indikator på närvaron av en kontur med `Math.abs(sx) + Math.abs(sy)`. Absolutbelopp behöver man eftersom man har negativa koefficienter i faltningsmatriserna.
4. Sätt pixeln till svart om indikatorn är större än tröskelvärdet, till vit annars. Mata in tröskelvärdet i Parameter-rutan.

Skriv en klass `SobelFilter` som implementerar denna algoritm.

Laboration 10 — ritprogram

Mål: Du ska lära dig mera om att skriva och använda klasser som är strukturerade med hjälp av arv. Du ska också lära dig att använda klassen `ArrayList` och att läsa data från fil.

Läsanvisningar

- Läroboken: avsnitt 9.1–9.6 (arv), 12.1–12.5 (`ArrayList`), 7.8 (läsning från fil).

Förberedelseuppgifter

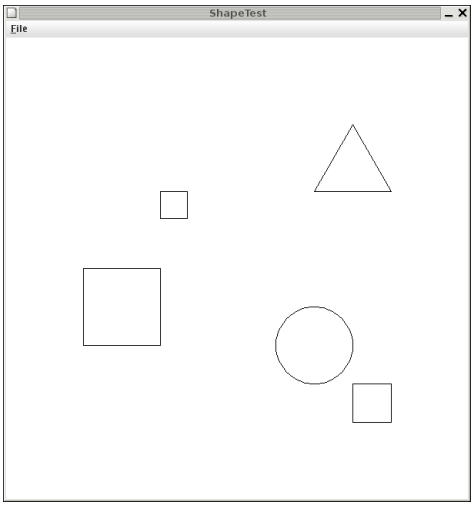
- Gör uppgift 1–3 för kvadrater, tänk igenom hur man kan rita trianglar och cirklar. Skissa på lösningen till uppgift 4 och 5.

Bakgrund

I ritprogram, till exempel `xfig`, kan man rita figurer som man sedan kan behandla på olika sätt. Man kan till exempel ändra storlek på figurerna, flytta dem och ta bort dem.

Här beskrivs ett extremt enkelt ritprogram. Det finns bara tre slags figurer: kvadrater, liksidiga trianglar och cirklar. Det enda användaren kan göra är att flytta omkring figurerna i ritfönstret genom att först klicka på en figur och sedan klicka på den nya positionen.

Ritfönstret har följande utseende:



I ritprogrammet beskrivs de gemensamma egenskaperna hos figurerna i en klass `Shape`. Denna klass används som superklass på klasserna som beskriver specifika figurer (kvadrat, triangel och cirkel).

```
public abstract class Shape {
    protected int x;
    protected int y;

    /** Skapar en figur med läget x,y */
    protected Shape(int x, int y) {
        this.x = x;
        this.y = y;
    }

    /** Ritar upp figuren i fönstret w */
    public abstract void draw(SimpleWindow w);
```

```
    /** Raderar bilden av figuren, flyttar figuren till newX,newY
        och ritar upp den på sin nya plats i fönstret w */
    public void moveToAndDraw(SimpleWindow w, int newX, int newY) {
        java.awt.Color savedColor = w.getLineColor();
        w.setLineColor(java.awt.Color.WHITE);
        draw(w);
        x = newX;
        y = newY;
        w.setLineColor(savedColor);
        draw(w);
    }

    /** Undersöker om punkten xc,yc ligger "nära" figuren */
    public boolean near(int xc, int yc) {
        return Math.abs(x - xc) < 10 && Math.abs(y - yc) < 10;
    }
}
```

I programmet håller man reda på figurerna som man skapar genom att man lägger in dem i en lista (ett objekt av klassen `ShapeList`). I följande program skapar man fem figurer, lägger in dem i listan och ritar upp dem:

```
import se.lth.cs.window.SimpleWindow;
import se.lth.cs.ptdc.shapes.ShapeList;

public class ShapeTest {
    public static void main(String[] args) {
        SimpleWindow w = new SimpleWindow(600, 600, "ShapeTest");
        ShapeList shapes = new ShapeList();
        shapes.insert(new Square(100, 300, 100));
        shapes.insert(new Triangle(400, 200, 100));
        shapes.insert(new Circle(400, 400, 50));
        shapes.insert(new Square(450, 450, 50));
        shapes.insert(new Square(200, 200, 35));
        shapes.draw(w);
    }
}
```

Parametrarna till konstruktorerna är figurens läge (x och y) och storlek (sidlängd eller radie). Klassen `ShapeList` har följande specifikation:

```
/** Skapar en tom lista */
ShapeList();

/** Lägger in figuren s i listan */
void insert(Shape s);

/** Ritar upp figurerna i listan i fönstret w */
void draw(SimpleWindow w);

/** Tar reda på en figur som ligger nära punkten xc,yc; ger null om
    ingen sådan figur finns i listan */
Shape findHit(int xc, int yc):
```

Operationen `findHit` används för att ta reda på vilken figur som användaren pekat och klickat på. Detta hanteras av klassen `CommandDispatcher`, som ansvarar för programmets kommunikation med användaren. Klassen har följande uppbyggnad:

```
import se.lth.cs.window.SimpleWindow;
import se.lth.cs.ptdc.shapes.Shape;
import se.lth.cs.ptdc.shapes.ShapeList;

class CommandDispatcher {
    private SimpleWindow w;
    private ShapeList shapes;

    public CommandDispatcher(SimpleWindow w, ShapeList shapes) {
        this.w = w;
        this.shapes = shapes;
    }

    public void mainLoop() {
        while (true) {
            // användaren klickar på en figur
            // användaren klickar på en ny position
            // figuren flyttas till den nya positionen
        }
    }
}
```

Uppgifter

Råd: i nedanstående uppgifter är det lämpligt att börja med uppgift 1 till 3 men bara hantera kvadrater, så att man ser att allt fungerar. Komplettera sedan programmet med cirklar och trianglar.

1. Klassen `Shape` finns färdigskriven (`se.lth.cs.ptdc.shapes.Shape`). Skriv tre subklasser till `Shape`: `Square`, `Triangle` och `Circle`. Lägg klasserna i separata filer i projektet *lab10* (*Square.java*, *Triangle.java*, *Circle.java*). I subklasserna måste du definiera vad som avses med ”läget” hos en figur. Koordinaterna `x` och `y` kan för en kvadrat ange övre vänstra hörnet, för en triangel nedre vänstra hörnet, för en cirkel medelpunkten. Man ska kunna bestämma figurens storlek med en parameter till konstruktorn.

   När man ska rita en cirkel är det enklast att tänka sig cirkeln som en regelbunden månghörning med många hörn. I lösningen till en av övningsuppgifterna i kapitel 9 i läroboken finns en metod som ritar en cirkel.

2. Klassen `ShapeTest` finns i filen *ShapeTest.java*. I klassen utnyttjas en färdigskriven version av klassen `ShapeList`. Tag bort kommentartecknen på `insert`-satserna och testkör programmet.

3. Skriv in klassen `CommandDispatcher`. Modifiera också `main`-metoden i `ShapeTest` så att ett `CommandDispatcher`-objekt skapas och operationen `mainLoop` utförs. Testkör det nya programmet.

4. Implementera klassen `ShapeList` med hjälp av en `ArrayList<Shape>` (ett programskelett finns i *ShapeList.java*). Testa programmet med din version av `ShapeList`. För att göra detta måste du ta bort `import`-satsen där `ShapeList` importeras från filerna *ShapeTest.java* och *CommandDispatcher.java*. Javasystemet kommer då att hitta den `ShapeList`-klass som finns i den aktuella katalogen, det vill säga din egen klass.

5. Ändra programmet så att man inte behöver använda `insert`-satser i `main`-metoden för att lägga in figurer i listan. I stället ska uppgifter om figurerna som ska skapas läsas från en

fil. I projektkatalogen finns en fil *shapedata.txt* med följande innehåll (filen specificerar samma figurer som i det tidigare exemplet):

```
S 100 300 100
T 400 200 100
C 400 400 50
S 450 450 50
S 200 200 35
```

Anmärkning: ett riktigt bra program skulle naturligtvis spara figurerna på filen, så att figurerna hade sina nya lägen nästa gång man körde programmet. Du får gärna lägga till denna funktionalitet i programmet.

6. Denna uppgift behöver du bara lösa om du har tid över. När man ska välja ut en figur att flytta känns det inte naturligt att man måste klicka på ett bestämt hörn (kvadrat och triangel) eller medelpunkten (cirkel). Det hade varit bättre om man kunde klicka på (eller nära) någon av figurens linjer. Detta kan man åstadkomma genom att skriva olika `near`-funktioner för de olika figurtyperna.

   Ändra ditt program så att fungerar på det beskrivna sättet. Det är lättast att skriva `near`-metoden i klassen `Circle`, så det är lämpligt att börja med den.

   Tips för kvadrater och trianglar: skriv en metod som undersöker om en punkt ligger nära en linje och anropa den med kvadratens fyra linjer och triangelns tre linjer. När man ska undersöka som en punkt ligger nära en sned linje är det enklast att räkna ut punktens avstånd till linjens båda ändpunkter och jämföra summan av dessa avstånd med linjens längd: