Programmering och databaser

DATORLABORATIONER

EDAA20

`http://cs.lth.se/edaa20`

LUNDS UNIVERSITET

Lunds Tekniska Högskola

2018/2019

EDAA20 Programmering och databaser Programmeringsteknik, godkända obligatoriska uppgifter 2018/2019

Skriv ditt namn och din namnteckning nedan:

Namn: .....

Namnteckning: .....

Godkända uppgifter		Datum	Handledarens namnteckning
Datorlaboration 1	introduktion		
Datorlaboration 2	textspel		
Datorlaboration 3	enkel-vektor		
Datorlaboration 4	anv-square		
Datorlaboration 5	jblockmole		
Datorlaboration 6	turtle		
Datorlaboration 7	maze		
Datorlaboration 8	textfil-polygon		
Datorlaboration 9	memory		
Datorlaboration 10	spelkort		
Datorlaboration 11	turtlerace-arv		

För att bli godkänd på en uppgift måste du lösa deluppgifterna och diskutera dina lösningar med en labhandledare. Denna diskussion är din möjlighet att få feedback på ditt program. Ta vara på den!

Se till att handledaren noterar dina uppgifter som godkända på detta godkännandeblad. Dessa underskrifter är ditt kvitto på att du är godkänd på laborationerna. Spara dem tills du fått slutbetyg i kursen.

Innehåll

Datorlaborationer – anvisningar

Datorlaborationerna innehåller lite större uppgifter och ger (tillsammans med övningsuppgifterna) viktig träning på det material som behandlas under kursen. De ska också ge praktisk träning på att skriva program på dator och examinera detta.

På laborationerna arbetar man normalt två och två. Det ger fördelen att man kan diskutera uppgifterna och hur de ska lösas med sin labbkompis. Men det medför också ett ansvar att arbeta på ett sätt att bägge lär sig så mycket som möjligt. Vill man kan man lösa en eller flera laborationer på egen hand.

I kursen ingår 11 obligatoriska laborationer. Det finns 14 labbtillfällen för detta. På dessa tillfällen finns det handledare på plats för att svara på dina frågor och hjälpa dig att komma vidare om du fastnat på någon uppgift. Handledarna ska också godkänna dina lösningar och ge dig feedback på din programkod. Anmälan till laborationsgrupp sker via kursens hemsida (`cs.lth.se/edaa20`).

I kursens veckoschema (finns på Moodle-sidan) framgår vilka laborationer du förväntas jobba med vilken vecka för att vara i fas. Du måste inte följa detta schema till punkt och pricka. Det gör alltså ingenting om du tillfälligt, t.ex. pga. sjukdoms, skulle släpa efter med en labb. Men det går inte att komma i slutet av kursen och redovisa många laborationer. De laborationer som inte är klara inom denna kursomgång får du redovisa vid nästa kurstillfälle (dvs. nästa läsår).

Du måste vara beredd att arbeta med laborationsuppgifterna mellan labbtillfälllena. Före laborationerna finns det förberedelseuppgifter som ska göras. Är du inte i fas får du jobba ikapp. Du får ansvara för balansgången att vara i fas och att inte stressa igenom laborationerna utan istället se till att du lär dig så mycket som möjligt av dem.

Tänk på att

- det inte är att du är godkänd på en laboration som är det viktiga utan vad du lärt dig på laborationen.
- man lär sig programmera genom att skriva program.
- inte glömma bort övningsuppgifterna (finns på Moodle-sidan). De ger också viktig träning.

Laboration 1 – introduktion

Mål: Under denna laboration ska du lära dig vad ett datorprogram är och se exempel på enkla program. Du ska också lära dig att editera, kompilera och exekvera enkla Java-program med hjälp av programutvecklingsverktyget Eclipse. Du ska också prova att använda Eclipse debugger.

Förberedelseuppgifter

- Läs avsnittet Bakgrund.
- Läs igenom anvisningarna om Eclipse (finns på kursens webbsida).

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

Detta är nästan samma program som beskrivs i Eclipse-handledningen, 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!”.

- Rad 1: en inläsningsklass `Scanner` importeras från ett ”paket” med namnet `java.util`. Detta paket och många andra ingår i Javas standardbibliotek.
- Rad 3: `public class Calculator` talar om att programmet, klassen, heter `Calculator`.
- Rad 4: `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 5: texten `Skriv två tal` skrivs ut på skärmen (`println` betyder ”print line”). När man använder Eclipse så hamnar utskrifter i konsolfönstret.
- Rad 6: 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 7: 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 8: samma sak med variabeln `nbr2`.
- Rad 9: variabeln `sum` deklareras. Talen `nbr1` och `nbr2` adderas och summan lagras i `sum`.
- Rad 10: resultatet (innehållet i variabeln `sum`) skrivs ut.

- Rad 11: slut på `main`-metoden.
- Rad 12: slut på klassen.

Uppgifter

1. Logga in på datorn.

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.

   Arbetsområdet, som heter edaa20-workspace, finns i packad form i filen edaa20-workspace.zip som hämtas från kursens hemsida:

   1. Starta en webbläsare och gå till `cs.lth.se/edaa20/ws`.
   2. En fil laddas automatiskt ner. Öppna den och packa upp innehållet. Kom ihåg var du lägger den.

      Nu har du fått en katalog edaa20-workspace. Katalogen innehåller bl.a. projektkatalogerna *cs_pt*, *Lab01*, *Lab02*,...
   3. Starta Eclipse. På Linux-maskinerna i datorsalarna kan du starta Eclipse genom att välja Eclipse (den senaste versionen) bland applikationerna. Fråga en av labbledarna ifall du inte hittar Eclipse. En dialogruta där du ska välja arbetsområde visas. Välj katalogen edaa20-workspace som tidigare skapades när du packade upp.

      I Eclipse-fönstret finns i projektvyn ("Package Explorer", längst till vänster) projekten *cs_pt*, *Lab01*, *Lab02*, *Lab03*, ... — det är de projektkataloger som finns i *edaa20-workspace*. Filerna med namn som börjar på *Lab* innehåller färdigskrivna program som utnyttjas under laborationerna. *cs_pt* innehåller en biblioteksfil (*.jar*-fil) med klasser som utnyttjas. I samma projekt finns källkoden (*.java*-filerna) till dessa klasser — de behövs inte för laborationerna, men en del brukar vara intresserade av att titta på dem.

3. 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 *Lab01* genom att klicka på plustecknet (eller pilen) bredvid projektet.

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

   c) Öppna paketet *(default package)* genom att klicka på plustecknet.

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

4. Kör programmet: markera *Calculator.java* i projektvyn, 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 programkod använder man decimalpunkt.

   Man kan köra det senaste programmet en gång till genom att klicka på Run-ikonen i verktygsraden.

5. 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.

6. 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*.

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

8. 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. (Läs avsnittet ”Hitta fel i program” i Eclipse-handledningen om du inte vet hur man gör.) 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 visas i ett av eclipse-fönsterna, se till att du ser detta och kan identifiera hur variablerna får sina värden under programmets exekvering.
   - 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.

9. Följande program använder den färdiga klassen `SimpleWindow`:

```
import se.lth.cs.pt.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);
```

```
    }
}
```

Dokumentationen av klassen `SimpleWindow` finns online via länk från kurswebbsidan. Förklaring av de viktigaste delarna av programmet:

- På den första raden importeras den färdiga klassen SimpleWindow.
- 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”. Referensvariabeln `w` tilldelas det nya fönsterobjektet.
- Sedan flyttas fönstrets ”penna” till punkten 100, 100.
- Slutligen ritas en linje.

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

a) Markera projektet *Lab01* i projektvyn,

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

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

d) Klicka på Finish.

Dubbelklicka på *SimpleWindowExample.java* för att ladda in filen i editorn (om detta inte redan gjorts automatiskt). Som du ser har Eclipse redan fyllt i 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.

10. 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.

11. Öppna filen *LineDrawing.java*. I filen finns följande kod, komplettera programmet så att det fungerar. Raderna med kommentarer ska ersättas med riktig programkod.

```
public class LineDrawing {
   public static void main(String[] args) {
       SimpleWindow w = new SimpleWindow(500, 500, "LineDrawing");
       w.moveTo(0, 0);
       while (true) {
           // vänta tills användaren klickar på en musknapp
           // rita en linje till den punkt där användaren klickade
       }
   }
}
```

Ledtråd: `SimpleWindow` innehåller också metoder för att ta hand om musklick. Dessa metoder 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();
```

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

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

Checklista

Exempel på vad du ska kunna efter laborationen:

- Starta Eclipse, öppna önskat arbetsområde (workspace) och öppna önskat projekt.
- Skapa .java-filer och skriva in enkla Java-program. (Med enkla program menas här ett program på några rader, som till exempel `Calculator`.)
- Kunna använda metoden `System.out.println` och förstå vad som händer när den används.
- Förstå vad det innebär att läsa in tal från tangentbordet.
- Spara .java-filer (kompilering sker då automatiskt).
- Exekvera program.
- Använda debuggern:
  - Sätta och ta bort brytpunkt.
  - Starta debuggern.
  - Köra fram till en brytpunkt med Resume.
  - Exekvera stegvis med Step over (och senare i kursen med Step Into).
  - Byta mellan debug-perspektivet och Java-perspektivet.

Laboration 2 – eget spel

Mål: Du ska skriva ett program med din egen kod. Du ska träna på att deklarera och använda variabler samt på att använda alternativ och repetition.

Förberedelseuppgifter

- Läs igenom texten i avsnittet Bakgrund.
- Hitta på ett spel och bestäm hur det ska fungera.
- Skissa ditt program (gärna pseudokod) på papper.

Bakgrund

På den här laborationen ska du skriva ett textbaserat spel. Du ska själv hitta på hur det ska fungera.

Inspiration

Här finns några förslag att hämta inspiration ifrån. Du kan använda något av förslagen, modifiera något av förslagen eller hitta på något helt eget spel.

- Spela ”gissa talet” och ge ledtrådar om talet är för litet eller för stort.
- Hitta på något enkelt tärningsspel att spela med användaren.
- Träna användaren i multiplikationstabellen.
- Rita många kvadrater i ett fönster och låta användaren gissa antalet.
- Kolla reaktionstiden hos användaren genom att mäta tiden det tar att trycka Enter efter att man fått vänta en slumpmässig tid på att strängen ”NU!” skrivs ut. Om man trycker Enter innan startutskriften ges blir den uppmätta tiden 0 och på så sätt kan ditt program detektera att användaren har tryckt för tidigt. Mät reaktionstiden upprepade gånger och beräkna medelvärdet.
- Låt användaren svara på flervalsfrågor om din favoritfilm.
- Spela sten/sax/påse med användaren.
- Låt användaren på tid så snabbt som möjligt skriva olika ord baklänges.

Några metoder som kan vara bra att ha:

- Dra slumptal:

  `Random`

```
/** Skapar en slumptalsgenerator(). */
Rendom();

/** Returnerar ett slumptal mellan 0 och bound-1. */
int nextInt(int bound);

/** Returnerar ett slumptal mellan 0.0 och 1.0. */
double nextDouble();
```

  Exempel:

```
Random rand = new Random();
int nbr = rand.nextInt(10); // nbr tilldelas slumptal mellan 0 och 9
```

  Om du använder någon av metoderna i klassen `Random` måste `Random` importeras. Lägg till följande rad i början av filen:

```
import java.util.Random;
```

- Mäta tid:

  `System`

```
/** Returnerar aktuell tid i millisekunder. */
static long currentTimeMillis()
```

  Exempel:

```
long start = System.currentTimeMillis();
```

- Låta programmet vänta en viss tid:

  `SimpleWindow`

```
/** Låter programmet vänta millis millisekunder. */
static void sleep(long millis);
```

  Exempel:

```
SimpleWindow.sleep(2000);   // Programmet väntar 2 sekunder
```

Krav på programmet

- Ditt program måste innehålla minst ett alternativ (`if`-sats) och minst en repetition (`for`- eller `while`-sats).
- Din kod ska vara lätt att läsa och förstå (bra val av variabelnamn, vettig indentering ...).
- Det är en konst att begränsa sig. Se till att du inte tar dig vatten över huvudet.
  - Kontrollera gärna din idé med labhandledaren i början av laborationen.
  - Utveckla ditt program stegvis. Börja med ett program som bara gör något litet. Då har du tidigt ett program som fungerar. Bygg sedan ut det efterhand.

Datorarbete

1. Logga in på datorn, starta Eclipse, öppna projektet *Lab02*.

2. Skapa en fil med namnet *SmallGame.java* (eller något namn du väljer själv). För att skapa en fil ska du
   - markera projektet *Lab02*
   - klicka på New Java Class-ikonen eller högerklicka och välja New och därefter Class
   - skriva namn på klassen (t.ex. `SmallGame`)
   - klicka på Finish

3. Skriv ditt program. Testa och se till att spelet fungerar som avsett.

4. Låt någon annan kursdeltagare läsa ditt program och köra spelet. Notera den återkoppling du får på programmet. På samma sätt ska du ge feedback på en annan kursdeltagares program.

5. Använd den återkoppling du fick för att förbättra ditt program.

Checklista

Exempel på vad du ska kunna efter laborationen:

- Skapa .java-filer och skriva enkla program i Eclipse.
- Skriva programkod med alternativ (`if`-sats).
- Skriva programkod med repetition (`for`- eller `while`-sats).

Laboration 3 – vektorer

Mål: Du ska träna på att läsa och felsöka programkod. Du ska träna på att skriva program där koden är uppdelad i metoder samt på vektorer.

Förberedelseuppgifter

- Läs igenom texten i avsnittet Bakgrund och lös den uppgift som finns där.

Bakgrund

SMHI har stora mängder data över Sveriges klimat och miljö. Följande värden över dygnsmedeltemperaturen i Lund under februari 2017 är hämtade från deras webbsida: 0.8, 0.3, 0.9, 2.4, 2.0, 0.8, -2.1, -2.6, -3.5, -0.6, -0.4, -0.9, -2.9, -2.4, -0.6, 2.0, 3.6, 2.9, 4.5, 5.7, 6.2, 5.7, 3.3, 0.4, -0.7, 5.3, 7.4, 5.5.

Vi kan bl.a. se att dygnsmedeltemperaturen var $0,8^\circ$ 1 februari och att dygnsmedeltemperaturen var under 0 den 7 februari.

SMHI har också definitioner för när det är vår, sommar etc.. I programmet nedan beräknas vilket datum det blev vår i Lund 2017. Studera programmet och svara på följande frågor:

- Vad skrivs ut (dvs. vilket datum startade våren)?
- Vad är det för kriterier som används i programmet för att definiera vilket datum våren startar.

```
public class ComputeFirstSpringDate {

    public static void main(String[] args) {
        double [] februaryTemp = {0.8, 0.3, 0.9, 2.4, 2.0, 0.8, -2.1, -2.6, -3.5, -0.6,
                                  -0.4, -0.9, -2.9, -2.4, -0.6, 2.0, 3.6, 2.9, 4.5, 5.7,
                                  6.2, 5.7, 3.3, 0.4, -0.7, 5.3, 7.4, 5.5,
                                  4.3, 3.1, 4.3, 3.8, 3.8, 1.2};
        System.out.println(firstSpringDate(februaryTemp));
    }

    /** Returnerar datumet för när våren startar.
        Om inte våren startar under månaden returneras -1.
        Vektorn a innehåller dygnsmedeltemperaturerna under månaden
        samt under de 6 första dygnen i nästkommande månad. */
    public static int firstSpringDate(double[] a) {
        int nbrDays = 0;
        for (int i = 14; i < a.length; i++) {
            if (a[i] > 0) {
                nbrDays++;
                if (nbrDays == 7) {
                    return i - 6 + 1;
                }
            } else {
                nbrDays = 0;
            }
        }
        return -1;
    }

}
```

Datorarbete

Felsökning

1. a. Logga in på datorn, starta Eclipse, öppna projektet *Lab03*.

   b. I den färdiga klassen `ArrayStatistics` finns metoder för att beräkna medelvärdet av talen i en vektor, minsta värdet etc. I klassen `MainTemp` finns en `main`-metod där det skapas en vektor med `double`-tal varefter metoderna i `ArrayStatistics` anropas. Det finns fem fel i metoderna i `ArraysStatistics`. Din uppgift är att hitta felen.

   - Provkör programmet. Blev utskrifterna de förväntade? Vad borde ha skrivits ut?
   - Sätt en brytpunkt på raden med första metodanropet. Kör programmet i debuggern. Om du klickar på `temp` i variabel-fönstret ser du vektorns innehåll.
   - Klicka på Step Into-ikonen för att ”gå in i metoden”. Stega sedan fram genom att klicka på Step over-för att se vad som händer i programmet och försök hitta felet och rätta det.
   - Gör på motsvarande sätt för att hitta de övriga felen. Om något exekveringsfel skulle inträffa så försök tolka felutskrifterna. Man kan bl.a. utläsa typen av fel och var i programmet felet inträffade (metod och radnummer).

Räkna antal förekomster

2. Vi kan ”kasta tärning” genom att dra slumptal i intervallet [1,6]. Metoden `nextInt` i klassen `Random` kan användas för detta. Slumptalen är rektangelfördelade, vilket innebär att alla tal i intervallet är lika sannolika. Om det stämmer ska vi få ungefär lika många 1:or, 2:or etc om vi kastar tärningen många gånger.

   Skriv ett program där du upprepade gånger kastar tärningen (drar ett slumptal i intervallet [1,6]). Programmet ska räkna och skriva ut andelen 6:or. I början av programmet ska antal kast läsas in så att du kan prova programmet med olika antal tärningskast.

   Kör programmet och kontrollera att resultatet blir rimligt. Resultatet bör närma sig $1/6$ om antal kast är stort.

3. Ändra programmet från förra uppgiften så att det räknar antal 1:or, 2:or, etc och sedan skriver ut de olika antalen. Nu behövs det 6 st räknare istället för en. Använd en vektor av typen `int[]` för räknarna. Du har två val: Antingen kan du skapa en vektor med 6 element och lagra antal 1:or på index 0, antal 2:or på index 1 osv. Eller också kan du skapa en vektor med 7 element och bara använda index 1 till 6.

   Anm. Detta sätt att använda en `int[]`-vektor för att lagra resultaten när man räknat antal förekomster av olika element passar bra när det är lätt att översätta element till index. I andra fall är det svårare, t.ex. om man ska räkna antal olika ord i en text. Då får man istället använda andra tekniker, t.ex. lagra ordet och antal förekomster tillsammans. Det finns färdiga klasser i Java (t.ex. `HashMap`) för att lagra par av element.

Checklista

Exempel på vad du ska kunna efter laborationen:

- Deklarera, skapa och använda vektorer med `double`- eller `int`-tal.
- Skriva program där koden är uppdelad i metoder.
- Använda debuggern som ett hjälpmedel att hitta fel i program.

Laboration 4 – använda färdigskrivna klasser, kvadrat

Mål: Du ska lära dig att läsa specifikationer av klasser och att utnyttja färdigskrivna klasser för att lösa enkla uppgifter. Du ska också lära dig lite mera om Eclipse.

Förberedelser

- Läs avsnittet Bakgrund.
- Tänk igenom följande frågor och se till att du kan svara på dem:
  1. Vad är en main-metod? Hur ser en klass med en main-metoden ut i stora drag?
  2. Vad är referensvariabler och vad har man dem till?
  3. Hur skapar man objekt?
  4. Hur anropar man en metod på ett objekt?
  5. Vad använder man parametrar till?

Bakgrund

En klass `Square` som beskriver kvadrater har nedanstående specifikation.

```
/** Skapar en kvadrat med övre, vänstra hörnet i x,y
    och med sidlängden side. */
Square(int x, int y, int side);

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

/** Raderar bilden av kvadraten i fönstret w. */
void erase(SimpleWindow w);

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

/** Tar reda på x-koordinaten för kvadratens läge. */
int getX();

/** Tar reda på y-koordinaten för kvadratens läge. */
int getY();

/** Tar reda på kvadratens area. */
int getArea();
```

Användning av klassen Square

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

```
1  import se.lth.cs.pt.window.SimpleWindow;
2  import se.lth.cs.pt.square.Square;
3
4  public class DrawSquare {
5      public static void main(String[] args) {
6          SimpleWindow w = new SimpleWindow(600, 600, "DrawSquare");
7          Square sq = new Square(250, 250, 100);
8          sq.draw(w);
```

```
9       }
10  }
```

- På rad 1 och rad 2 importeras de klasser som behövs, i detta fall: klassen `SimpleWindow` och klassen `Square`. De klasserna finns inte i Javas standardbibliotek utan har skrivits speciellt för kurserna i programmeringsteknik. 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`.
- På rad 6 skapas ett `SimpleWindow`-objekt. Refererensvariabeln `w` refererar till detta objekt.
- Därefter skapas ett `Square`-objekt som referensvariabeln `sq` refererar till.
- Slutligen ritas kvadraten `sq`.

Notera parametrarna som man använder när man skapar objekten. I `new`-uttrycket som skapar kvadratobjektet står det till exempel (250, 250, 100). Det betyder att kvadraten ska ha läget 250, 250 och sidlängden 100. Läget och sidlängden kan man ändra senare i programmet, med `sq.move(dx,dy)` och `sq.setSide(newSide)`.

Lägg också märke till att referensvariablerna `w` och `sq` har olika typer. En referensvariabels typ avgör vilka slags objekt variabeln får referera till. `w` får referera till `SimpleWindow`-objekt och `sq` får referera till `Square`-objekt.

Användning av klassen SimpleWindow

Bläddra fram källkoden till klassen `Square`. Du hittar klassen i Eclipse-projektet *cs_pt* under katalogen *src* och i paketet `se.lth.cs.pt.square`. Titta på källkoden och försök förstå vad som händer. Några av metoderna i `SimpleWindow` (metoderna för att flytta pennan och för att rita linjer) utnyttjas i metoden `draw` i klassen `Square`.

`SimpleWindow` innehåller också metoder, som inte används i `Square`, t.ex. för att ta hand om musklick. Dessa metoder 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.pt.window.SimpleWindow;

public class PrintClicks {
    public static void main(String[] args) {
        SimpleWindow w = new SimpleWindow(600, 600, "PrintClicks");
        while (true) {
            w.waitForMouseClick();
            w.moveTo(w.getMouseX(), w.getMouseY());
            w.writeText("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 mellan punkten och den förra punkten som användaren klickade på. Vi sparar hela tiden koordinaterna för den förra punkten (variablerna `oldX` och `oldY`).

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

```
import se.lth.cs.pt.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 x = w.getMouseX();
            int y = w.getMouseY();
            w.moveTo(x, y);
            int xDist = x - oldX;
            int yDist = y - oldY;
            w.writeText("Avstånd: " + Math.sqrt(xDist * xDist + yDist * yDist));
            oldX = x;
            oldY = y;
        }
    }
}
```

Datorarbete

1. Logga in på datorn, starta Eclipse, öppna projektet *Lab04*. (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/edaa20` . Under Dokumentation finns en länk till dokumentationen av de färdigskrivna klasser som används under laborationerna, alltså klasserna i paketet `se.lth.cs.pt`. Leta upp och titta igenom specifikationen av klassen `Square`. Det är samma specifikation som finns under Bakgrund, fast i något annorlunda form.

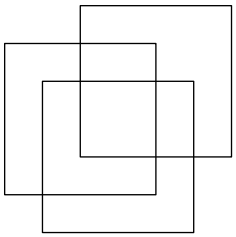
3. Klassen `DrawSquare` finns i filen *DrawSquare.java*. Öppna filen, kör programmet.

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 *(default package)*, 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:

   Testa programmet, rätta eventuella fel.

5. Någonstans i ditt program skapas ett kvadratobjekt. Det görs i en sats som har ungefär följande utseende: `Square sq = new Square(300,300,200)`. Ta 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 argumentet `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.

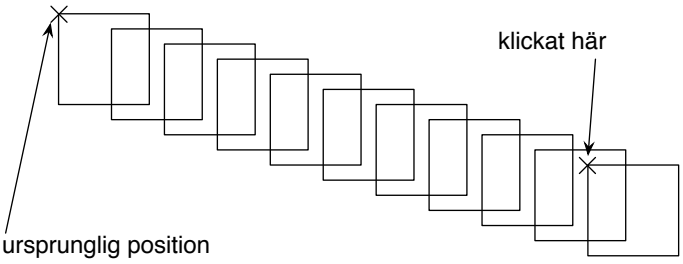
6. Kör programmen `PrintClicks` och `PrintClickDists`.

7. 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. Titta på dokumentationen av SimpleWindow och Square som finns länkad från kurshemsidan. Vilka metoder som verkar användbara för uppgiften kan du hitta där?

   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, klicka på Finish) eller kopiera någon av de filer som du redan har.

   Testa programmet.

8. Modifiera programmet i uppgift 7 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.

9. Observera att programmet från uppgift 8 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() - 10);
}
```

Kodsnutten ovan animerar en ny mindre kvadrat tills dess att kvadratens sida blivit mindre än eller lika med noll.

Kopiera koden från uppgift 8 till en ny fil `AnimatedSquare.java`. Lägg in fördröjning och radering så att kvadratbilden blir ”animerad”. (Använd *inte* `SimpleWindow`-metoden `clear`.)

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.

10. 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. Två kommandon som man ofta använder:

- 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. Observera att programmet ska vara korrekt formaterat när du visar det för labhandledaren för att få det godkänt. Notera också kortkommandot som står intill menyalternativet. Prova att använda kortkommandot också, och lär dig gärna det, då det är snabbare och mer bekvämt än att klicka.

- 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).

11. Kontrollera med hjälp av checklistan nedan att du behärskar de olika momenten i laborationen. Diskutera med övningsledaren om någonting är oklart.

Checklista

Exempel på vad du ska kunna efter laborationen:

- Tyda specifikationer av klasser.
- Skriva program som använder färdiga klasser:
  - Deklarera referensvariabler och skapa objekt.
  - Anropa metoder på objekt.

Laboration 5 – blockmullvad

Mål: Du ska lära dig mer om att strukturera kod genom att skriva och använda egna klasser. Du får öva på att skriva metoder och deklarera attribut. Du får även använda statiska variabler.

Förberedelseuppgifter

- Påbörja gärna datorarbetet nedan.

Bakgrund

Blockmullvad (*Talpa laterculus*) är ett fantasidjur i familjen mullvadsdjur. Den är känd för sitt karaktäristiska kvadratiska utseende. Den lever mest ensam i sina underjordiska gångar som till skillnad från mullvadens (*Talpa europaea*) har helt raka väggar.

Datorarbete

1. Du ska själv bygga upp ditt Java-program steg för steg i Eclipse men projektet finns förberett i det workspace som använts på tidigare laborationer.

   a) Skapa filen `Mole.java` genom att:

      a) Högerklicka på projektet *Lab05* i projektvyn,
      b) Expandera New i menyn och välj Class.
      c) Skriv namnet på klassen (`Mole`).
      d) Klicka på Finish.

   b) Öppna din nya klass (om det inte redan gjorts automatiskt). Lägg till en `main`-metod i klassen som skriver ut texten *Keep on digging!* med hjälp av `System.out.println`.

   c) Testa att köra ditt program. Om allt går bra ska texten du angivit skrivas ut i konsolfönstret i Eclipse.

   d) Nu har du skrivit ett Java-program som skriver ut en uppmaning till en mullvad att fortsätta gräva. Det programmet är inte så användbart, eftersom mullvadar inte kan inte läsa. Nästa steg är att skriva ett grafiskt program, snarare än ett textbaserat.

   Funktionen `println` som anropas i `main`-funktionen ingår i Javas standardbibliotek. Ett programbibliotek innehåller kod som kan användas av andra program, och för de flesta programspråk ingår ett standardbibliotek som alla program kan nyttja. Till grafiken i denna uppgift ska du, precis som i tidigare laborationer, använda den färdiga klassen `SimpleWindow`. Den finns i biblioteket *cs_pt* i ditt workspace.

   e) Låt `main`-metoden innehålla följande satser:

```
SimpleWindow w = new SimpleWindow(300, 500, "Digging");
w.moveTo(10, 10);
w.lineTo(10, 20);
w.lineTo(20, 20);
w.lineTo(20, 10);
w.lineTo(10, 10);
```

   Överst i filen måste du ha raden

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

   för att klassen `SimpleWindow` ska bli tillgänglig.

Koden känner du nog igen från tidigare i kursen. Den första raden skapar ett nytt `SimpleWindow` som ritar upp ett fönster som är 300 bildpunkter brett och 500 bildpunkter högt med titeln *Digging*. `SimpleWindow` har en *penna* som kan flyttas runt och rita linjer. Anropet `w.moveTo(10, 10)` flyttar pennan för fönstret `w` till position $(10, 10)$ utan att rita något, och anropet `w.lineTo(10, 20)` ritar en linje därifrån till position $(10, 20)$.

f) Kör ditt program. Du ska nu få upp ett fönster med en liten kvadrat utritad i övre vänstra hörnet.

2. Hela ditt program är för tillfället samlat i en och samma metod, vilket fungerar bra för väldigt små program. Nu ska vi strukturera om programmet så det blir lättare att utöka senare.

a) Skapa en ny klass med namnet `Graphics` (se punkt a i föregående uppgift om du glömt hur man skapar en klass) och flytta dit deklarationen av fönstret `w` så att det blir ett attribut i klassen. Skapa en ny metod i klassen `Graphics` med namnet `square`, och flytta dit koden som ritar kvadraten.

Filen `Graphics.java` ska se ut såhär:

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

public class Graphics {
    private SimpleWindow w = new SimpleWindow(300, 500, "Digging");

    public void square(){
        // Fyll i koden för att rita en kvadrat här.
        // Observera att w är definerat ovan.
    }
}
```

Nu har vi beskrivit vad klassen `Graphics` ska innehålla och hur den ska fungera. Men för att faktiskt använda den måste vi göra något mer. Vi måste anropa metoden `square` i `main`-metoden. Metoden `square` finns i klassen `Graphics`, och därför måste ett `Graphics`-objekt skapas först. Därefter använder vi punktnotation `square`-metoden genom att med punktnotation ange att det är en metod inuti `Graphics`-objektet som anropas.

Filen `Mole.java` ska se ut såhär:

```
public class Mole {

    public static void main(String[] args) {
        Graphics g = new Graphics();
        g.square();
    }

}
```

b) Kör programmet, om allt fungerar ska programmet göra samma sak som i föregående uppgift.

c) Kommentera bort raden `g.square();` genom att sätta `//` framför. Kör programmet igen och se vad som händer. Varför är det så?

d) Ta bort kommentartecknen vid `g.square();` igen och kommentera nu istället bort raden `Graphics g = new Graphics();`. Vad händer då och varför? Diskutera med

din labhandledare om det känns oklart. Ta bort kommentaren och kontrollera att programmet fungerar innan du går vidare.

3. Nu ska du skapa ett nytt koordinatsystem för `Graphics` som har *stora* bildpunkter. Vi kallar `Graphics` stora bildpunkter för *block* för att lättare skilja dem från `SimpleWindows` bildpunkter. Om blockstorleken är b, så ligger koordinaten (x, y) i `Graphics` på koordinaten (bx, by) i `SimpleWindow`.

   a) Lägg till följande deklarationer som attribut överst i klassen `Graphics`.

   ```
   private int width;
   private int blockSize;
   private int height;
   ```

   b) Nu vill vi att våra attribut ska få startvärden. Ett heltalsattribut är noll om inget annat anges. Om man vill att ett attribut alltid ska få samma startvärde så går det att tilldela värdet direkt vid deklarationen (alltså t.ex. `private int width = 30;`). Men i denna laborationen ska vi öva på att ge attributen dess startvärden via konstruktorn. På så vis kan den som skapar objektet själv välja vilka värden attributen ska få.

   En konstruktor har alltid samma namn som klassen. Skapa en konstruktor med heltalsparametrarna `w`, `h` samt `bs`. Låt värdet av `w` tilldelas till attributet `width`, värdet av `h` tilldelas till attributet `height` och värdet av `bs` tilldelas till attributet `blockSize`.

   c) Ändra bredden på ditt `SimpleWindow` till `width * blockSize` och ändra höjden till `height * blockSize`. Detta betyder att fönstret måste skapas i konstruktorn, men det måste ändå deklareras som attribut. Nu ska din klass alltså ha följande struktur:

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

   public class Graphics {
       private int width;
       private int blockSize;
       private int height;

       private SimpleWindow w;

       public Graphics(int w, int h, int bs){

           // Lägg till satser för att initiera width, blocksize och height

           this.w = new SimpleWindow(width * blockSize,
                                     height * blockSize,
                                     "Digging");
       }

       // Metoden square som du skrivit tidigare
   }
   ```

   d) Nu har du ändrat i `Graphics`. Bland annat har du ändrat hur objekt av klassen skapas. Detta kräver att du gör motsvarande ändring i klassen `Mole`. Nu måste du skicka in värden på varje parameter (`w`, `h` och `bS`) – dessa värden kallas argument. Ändra därför så att `main` nu ser ut så här i stället:

   ```
   public class Mole {
       public static void main(String[] args) {
           Graphics g = new Graphics(30,50,10);
           g.square();
       }
   }
   ```

Provkör och kontrollera att programmet fortfarande fungerar som innan.

e) Innan vi gjorde en egen konstruktor kunde vi ändå skapa ett nytt `Graphics`-objekt, men då utan att ange några värden. Varför? (Hitta ledtrådar genom att googla ”Java default constructor”).

f) Skapa en ny metod i `Graphics` med namnet `block` och två parametrar `x` och `y` av typen `int` och returtypen `void`. Metodens *kropp* ska se ut såhär:

```
int left = x * blockSize;
int right = left + blockSize - 1;
int top = y * blockSize;
int bottom = top + blockSize - 1;
for(int row = top; row <= bottom; row ++){
    w.moveTo(left, row);
    w.lineTo(right, row);
}
```

g) Metoden `block` ritar ett antal linjer. Hur många linjer ritas ut? I vilken ordning ritas linjerna?

h) Anropa funktionen `block` några gånger i `main`-metoden så att några olika block ritas upp i fönstret när programmet körs. Kör ditt program och kontrollera resultatet.

4. Det finns många sätt att beskriva färger. I naturligt språk har vi olika namn på färgerna, till exempel *vitt*, *rosa* och *magenta*. I datorn är det vanligt att beskriva färgerna som en blandning av *rött*, *grönt* och *blått* i det så kallade RGB-systemet. `SimpleWindow` använder typen `java.awt.Color` för att beskriva färger och `java.awt.Color` bygger på RGB. Det finns några fördefinierade färger i `java.awt.Color`, till exempel `java.awt.Color.BLACK` för svart och `java.awt.Color.GREEN` för grönt. Andra färger kan skapas genom att ange mängden rött, grönt och blått.

a) Skapa en ny klass med namnet `Colors` och lägg in följande definitioner:

```
public static final Color MOLE = new Color(51, 51, 0);
public static final Color SOIL = new Color(153, 102, 51);
public static final Color TUNNEL = new Color(204, 153, 102);
```

För att använda klassen `java.awt.Color` är det enklast att importera den genom att skriva `import java.awt.Color;` överst i filen.

Den tre parametrarna till `new Color(r, g, b)` anger hur mycket *rött*, *grönt* respektive *blått* som färgen ska innehålla, och mängderna ska vara i intervallet 0–255. Färgen $(153, 102, 51)$ innebär ganska mycket rött, lite mindre grönt och ännu mindre blått och det upplevs som brunt. Klassen `Colors` fungerar här som en färgpalett, men vi har inte ritat något med färg ännu. Kompilera och kör ditt program ändå, för att se så programmet fungerar likadant som sist.

b) Lägg till en parameter till metoden `block` i klassen `Graphics`. Skriv dit den sist i parameterlistan med namnet `color` och typen `java.awt.Color`. För att ändra färgen på blocket, till den färg som angivits som parameter, ska du byta linjefärg innan du ritar. Lägg till följande rad i början av metoden `block`:

```
w.setLineColor(color)
```

c) Ändra i `main` och lägg till en av färgerna från klassen `Colors` som tredje argument i dina anrop till `block`. Eftersom de färger du har definerat i ditt program

är publika (`public`) och statiska (`static`) i klassen Colors behöver vi denna gången inte skapa något objekt. Ett anrop till metoden `block` kan därför se ut så här: `g.block(10,2,Colors.MOLE);` Kompilera och kör ditt program och upplev världen i färg.

Ytterligare förklaring: Normalt sett har varje objekt sina egna värden på varje attribut (som i fallet med `width`, `height`, `blockSize` och `w` i klassen `Graphics`) men `static` betyder att det bara finns ett värde i hela programmet (det är alltså inte unikt per objekt). `public` betyder att det går att komma åt värdena även från andra klasser och final betyder att det aldrig kan ändras under programmets gång. Detta betyder att vi har definierat färgerna som tre globala konstanter. Konventionen i Java är att konstanter namnges med enbart versaler. Punktnotation används alltid för att komma åt saker i en annan klass eller ett annat objekt, och därför skriver vi `Colors.MOLE` för att komma åt färgen.

Tips: undvik att ange attribut som `public` eller `static` om du inte är riktigt säker på din sak. De flesta attribut framöver i kursen är privata och icke-statiska.

d) Eftersom attributen i klassen `Graphics` är privata kan vi inte komma åt dessa från klassen `Mole`. Men ofta finns det behov av att i efterhand komma åt attributens värden. I nästa uppgift kommer vi behöva komma åt bredden och höjden därför ska vi nu implementera get-metoder för dessa attribut.

En get-metod har ofta samma namn som attributet men med `get` framför. Då en metod består av flera ord låter man första ordet börja med liten bokstav (eftersom metoder alltid ska börja med liten bokstav) och alla efterföljande ord börja med stor bokstav (för att det ska bli lättare att läsa). Vidare brukar varje get-metod enbart returnera attributets värde, det betyder att get-metodernas returtyp ska vara samma som attributets typ. Det leder oss fram till att vi behöver följande i klassen `Graphics`:

```
public int getWidth() {
    return width;
}

public int getHeight() {
    return height;
}
```

Fråga din handledare om du behöver ytterligare förklaring.

5. Nu ska du skriva en funktion för att rita en rektangel och rektangeln ska ritas med hjälp av funktionen `block`. Sen ska du rita upp mullvadens underjordiska värld med hjälp av denna funktion.

a) Lägg till en metod i klassen `Graphics` med namnet `rectangle`. Metoden ska ta fem parametrar: `x`, `y`, `width` och `height` av typen `int`, samt `c` av typen `Color`.

Parametrarna `x` och `y` anger `Graphics`-koordinaten för rektangelns övre vänstra hörn och `width` och `height` anger bredden respektive höjden på rektangeln. Använd följande `for`-satser för att rita ut rektangeln.

```
for (int yy = y; yy < y + height; yy++){
    for(int xx = x; xx < x + width; xx++){
        block(xx, yy, c);
    }
}
```

b) I vilken ordning ritas blocken ut?

c) Skriv en metod i klassen `Mole` med namnet `drawWorld` som ritar ut mullvadens värld, det vill säga en massa jord där den kan gräva sina tunnlar. `drawWorld` ska inte ha några parametrar, returtypen ska vara `void`, och metoden ska anropa `rectangle` för att rita en rektangel med färgen `Colors.SOIL` som precis täcker fönstret. Anropa `drawWorld` i `main`-metoden och testa så att det fungerar genom att köra programmet.

Ledning: Eftersom funktionen `rectangle` finns i klassen `Graphics` så måste vi i klassen `Mole` ha tillgång till det `Graphics`-objekt som ska användas. Vi lägger det därför som attribut i klassen Mole. Eftersom attributet och metoden `drawWorld` inte är statiska måste vi då skapa ett objekt av typen `Mole` för att sen kunna använda metoderna. (Exekveringen startar ju i `main`, som är statisk, därför finns inget `Mole`-objekt från början, trots att vi är inuti klassen.) `Mole` måste alltså att se ut så här:

```
public class Mole {
    private Graphics g = new Graphics(30, 50, 10);

    public static void main(String[] args) {
        Mole m = new Mole();
        m.drawWorld();
    }

    public void drawWorld(){
        // Kod som ritar upp världen med hjälp av g.rectangle...
    }
}
```

Eftersom vi inte definerat någon konstruktor i klassen `Mole` är det återigen default-konstruktorn som anropas (vilket är tillräckligt för denna uppgift eftersom inga speciella värden måste tilldelas i konstruktorn).

6. I `SimpleWindow` finns en metod för att känna av tangenttryckningar. Du ska använda denna för att styra en liten blockmullvad.

a) Lägg till följande metod i klassen `Graphics`:

```
public char waitForKey() {
    return w.waitForKey();
}
```

I `Graphics`-klassen finns alltså nu en metod för att invänta en tangenttryckning. Den har vi implementerat genom att anropa metoden med samma namn i vårt `SimpleWindow`-objekt `w`.

Man kan säga att klassen `Graphics` *delegerar* uppgiften till `SimpleWindow`. Ser du varför det är en passande term?

7. Nu är det dags att få mullvaden att gräva på riktigt.

   a) Lägg till en funktion i klassen `Mole` med namnet `dig`, utan parametrar och med returtypen `void`. Funktionens kropp ska se ut såhär (fast utan `/* TODO */`):

   ```
   int x = g.getWidth() / 2;    // För att börja på mitten
   int y = g.getHeight() / 2;
   while (true) {
       g.block(x, y, Colors.MOLE);
       char key = g.waitForKey();

       if (key == 'w') { /* TODO */ }
       else if (key == 'a') { /* TODO */ }
       else if (key == 's') { /* TODO */ }
       else if (key == 'd') { /* TODO */ }
   }
   ```

   Byt ut i alla `/* TODO */` mot Java-satser så att `'w'` styr mullvaden ett steg uppåt, `'a'` ett steg åt vänster, `'s'` ett steg nedåt och `'d'` ett steg åt höger.

   b) Ändra `main` så att den anropar både `drawWorld` och `dig`. Kompilera och kör ditt program för att se om programmet reagerar på knapptryck på w, a, s och d.

   c) Om programmet fungerar kommer det bli många mullvadsfärgade block som tillsammans bildar en lång mask, och det är ju lite underligt. Lägg till ett anrop i `dig` som ritar ut en bit tunnel på position (x, y) efter anropet till `waitForKey` men innan `if`-satserna. Kompilera och kör ditt program för att gräva tunnlar med din blockmullvad.

Frivilliga extrauppgifter

8. Mullvaden kan för tillfället gräva sig utanför fönstret. Lägg till några `if`-satser i början av `while`-satsen som upptäcker om `x` eller `y` ligger utanför fönstrets kant och flyttar i så fall tillbaka mullvaden precis innanför kanten.

9. Mullvadar är inte så intresserade av livet ovanför jord, men det kan vara trevligt att se hur långt ner mullvaden grävt sig. Lägg till en himmelsfärg och en gräsfärg i objektet `Colors` och rita ut himmel och gräs i `drawWorld`. Justera också det du gjorde i föregående uppgift, så mullvaden håller sig under jord. (*Tips:* Den andra parametern till `Color` reglerar mängden grönt och den tredje parametern reglerar mängden blått.)

10. Ändra så att mullvaden kan springa uppe på gräset också, men se till så att ingen tunnel ritas ut där.

Checklista

I den här laborationen har du övat på att

- Bygga upp ett program steg för steg och testa ofta
- Strukturera vårt program med klasser och metoder
- Skriva egna klasser och metoder
- Skapa och anropa en egen konstruktor
- Anropa default-konstruktorn
- Definera och använda statiska konstanter
- Skriva get-metoder

Laboration 6 – implementera klasser, Turtle

Mål: Du ska fortsätta att träna på att implementera och använda klasser. Du ska också få mer träning i att använda Eclipse debugger.

Förberedelseuppgifter

- Tänk igenom följande frågor och se till att du kan svara på dem:
  1. Vad är en specifikation respektive en implementering av en klass?
  2. Vad är ett attribut? Var deklareras attributen? När reserveras det plats för dem i datorns minne och hur länge finns de kvar?
  3. När exekveras satserna i konstruktorn? Vad brukar utföras i konstruktorn?
  4. Vilka likheter/skillnader finns det mellan attribut, lokala variabler och parametrar.
  5. Vad menas med public och private?
- Läs avsnittet Bakgrund.

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 axlarna:

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

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

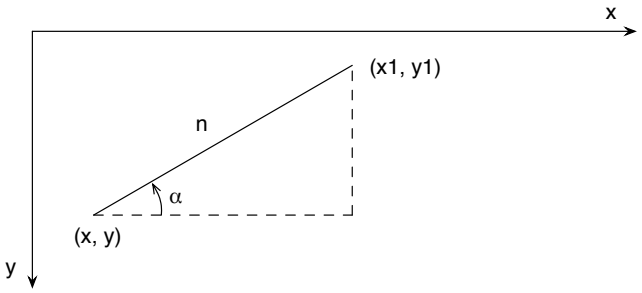
För att dra slumptal måste man skapa ett `Random`-objekt och att man får ett nytt slumpmässigt heltal i intervallet [0, n) med funktionen `nextInt(n)`. Skrivsättet [0, n) betyder att 0 ingår i intervallet, n ingår inte i intervallet. `rand.nextInt(100)` ger alltså ett slumpmässigt heltal mellan 0 och 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`. booleanvariabler kan bara anta två värden: `true` eller `false`. Man testar om en booleanvariabel `isPenDown` har värdet `true` med `if (isPenDown) ...`.

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 metoden `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));
```

Datorarbete

1. I filen *Turtle.java* i projektet *Lab06* finns ett ”skelett” (en klass utan attribut och med tomma metoder) till klassen `Turtle`.

   Implementera 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 de övriga metoderna.

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

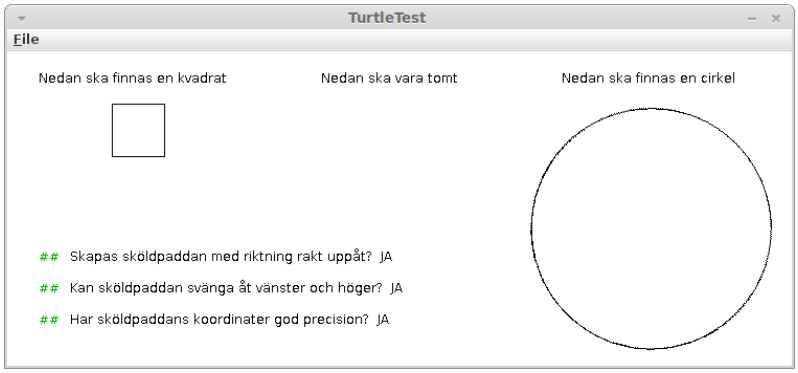
   Använd gärna debuggern för att hitta eventuella svårfunna fel. Här kan det vara bra att utnyttja kommandot Step Into. Det fungerar som Step Over med skillnaden att man följer exekveringen in i metoder som anropas.

3. Du ska nu träna mer på att använda debuggern i Eclipse. I fortsättningen förutsätter vi att du utnyttjar debuggern för att hitta fel i dina program.

   Välj ett av programmen från uppgift 2. Sätt en brytpunkt på en rad där en metod i Turtle anropas, t. ex. `t.penDown()` eller något liknande. Kör programmet i debuggern. Använd Step Into och följ hur man från `main`-metoden ”gör utflykter” in i metoderna i klassen `Turtle`.

Lägg märke till vilka storheter (variabler, attribut, parametrar) som är tillgängliga när man ”är i” `main`-metoden resp. inuti någon metod i klassen `Turtle`. **Tips!** Om du klickar på trekanten framför this i variabelvyn visas `Turtle`-objektets attribut.

4. I projektet finns också programmet `TurtleTest`, som testar din `Turtle`-klass i olika avseenden. Ett antal misstag kan upptäckas på detta sätt. Resultatet ska se ut så här:



Kör programmet `TurtleTest`. Om de ritade figurerna är felaktiga, eller något av testerna besvaras med ”NEJ”, gå tillbaka till din `Turtle`-klass och åtgärda felet.

5. 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 $[-180, 180]$.

6. 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).

7. Betrakta klassen `Turtle`. Ge exempel på ett attribut, en parameter och en lokal variabel. Ange också var i klassen/programmet respektive storhet är tillgänglig och får användas.

Exempel på	*Namn*	*Får användas var*
Attribut		
Parameter		
Lokal variabel		

Checklista

I den här laborationen har du övat på

- skillnaden mellan attribut, parametrar och lokala variabler,
- att använda konstruktorer för att sätta attributens startvärden,
- att använda klassen Math för beräkningar, och
- att skapa och använda en egen klass.

Laboration 7 – algoritmer, labyrint

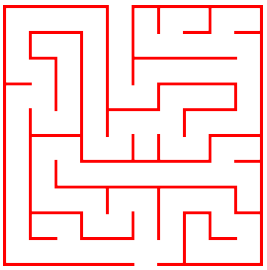
Mål: Du ska lösa ett problem där huvuduppgiften är att konstruera en algoritm för att hitta vägen genom en labyrint. Det ska du göra genom att skriva program som använder din `Turtle`-klass från laboration 6.

Förberedelseuppgifter

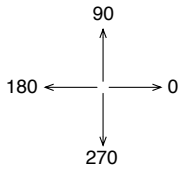
- Läs avsnittet Bakgrund.

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 nedersta väggen, och utgången alltid i den översta. 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:



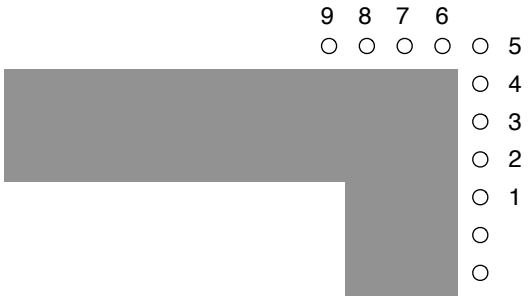
Labyrinten beskrivs av en färdigskriven klass `Maze`. Labyrintens utseende läses in från en fil när labyrintobjektet skapas. Klassen har följande specifikation:

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

Väggarna ritas med lite tomrum mellan sköldpaddan och väggen, så att det blir lättare att skilja dem åt. 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 `MazeWalker` som låter en sköldpadda vandra i en labyrint. Låt klassen ha följande uppbyggnad:

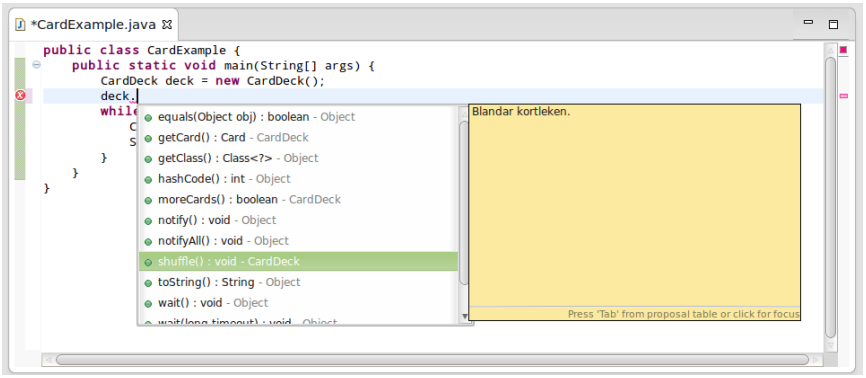
```
public class MazeWalker {
    private Turtle turtle;

    public MazeWalker(Turtle turtle) {
        // fyll i kod
    }

    /** Låter sköldpaddan vandra genom labyrinten maze, från
        ingången till utgången. */
    public void walk(Maze maze) {
        // fyll i kod
    }
}
```

Uppgifter

1. (valfritt) Om du vill kan du få lite mer hjälp av Eclipse när du programmerar. Ta fram fönstret med Eclipse-inställningarna:

   Window → *Preferences* (Studentdatorerna, Windows, Linux)
   Eclipse → *Inställningar...* (Mac)

   Välj *Java* → *Editor* → *Content Assist* → *Advanced*. Klicka i *Java proposals* och *Template proposals*. Klicka *Apply*, därefter *OK*. Hädanefter kommer Eclipse att ge förslag beroende på sammanhanget. Följande bild visar hur det kan se ut när programmeraren skrivit "`deck`", tryckt punkt, och nu bläddrar bland förslagen:



2. Till denna uppgift finns inte något färdigt Eclipseprojekt. Du måste alltså skapa ett sådant:

   1. Klicka på *New*-ikonen i verktygsraden. Välj *Java Project*.
   2. Skriv namnet på projektet (t.ex. *MazeTurtle*).
   3. Klicka på *Finish*.

3. Programmet som du skriver kommer att använda klasserna `SimpleWindow och Maze`. Dessa klasser finns i biblioteksfilen *cs_pt.jar*, som finns i projektet *cs_pt*. Programmet kommer också att utnyttja klassen `Turtle` från laboration 6.

   Du måste därför göra *cs_pt.jar* och projektet *Lab06* tillgängliga i ditt nya projekt.

   1. Högerklicka på projektet *MazeTurtle*, välj *Build Path* → *Configure Build Path*.
   2. Klicka på fliken *Libraries*. Om det finns en rad *Classpath*, markera då den.
   3. Klicka på *Add JARS...*, öppna projektet *cs_pt*, markera filen *cs_pt.jar* och klicka *OK*.
   4. Klicka nu på fliken *Projects* (istället för *Libraries*). Även här markerar du raden *Classpath* (om den finns).
   5. Klicka på *Add...*, markera *Lab06* och klicka *OK*.
   6. Klicka på *OK* för att lämna dialogen.

4. Skapa klassen `MazeWalker`. 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. (Använd t.ex. metoden `delay` i `SimpleWindow`; ett exempel finns på s. 20 i detta kompendium.)

5. 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, numrerade 1–5. Givetvis ska alla testas och fungera. För den sista labyrinten, som är ganska stor, kan du behöva justera (eller kanske helt ta bort) fördröjningen mellan varje steg.

Frivilliga extrauppgifter

6. Utforska källkoden för `Maze` som finns i projektet *cs_pt*. Kan du se hur du kan använda en egen karta[1] för labyrinten? (Tips: undersök `Maze`-klassens konstruktorer.)

7. Summera antalet steg sköldpaddan tar och antalet riktningsändringar som sköldpaddan gör och skriv ut dessa summor när sköldpaddan kommit i mål.

[1] Du kan exempelvis skapa en egen karta på `http://www.mazegenerator.se`. Klicka i rutan ”Använd alltid PNG”, välj ”Skapa ny”, högerklicka på bilden och spara den. Om du använder en egen bild måste den ha helvit eller transparent bakgrund.

Laboration 8 – spara data på textfil, polygoner

Mål: Du ska skriva program som läser data från en textfil. Samtidigt får du träning på att använda vektorer med objekt.

Förberedelseuppgifter

- Läs igenom texten i avsnittet Bakgrund.

Bakgrund

Hittills i kursen har du skrivit program med indata som användaren skriver i konsolfönstret. Resultat från program har du skrivit ut i konsolfönstret. Ofta behöver man spara data mellan exekveringarna. Detta kan göras på olika sätt, t.ex. kan program vara kopplade till stora databaser där data lagras i olika tabeller. På den här labben ska vi välja en enklare lösning och spara data på en textfil.

En textfil är en fil som är läsbar och som kan redigeras med en vanlig textredigerare. Exempel på textfiler är de *.java*-filer som innehåller den programkod du skrivit. De filer som producerats av java-kompilatorn (med tillägget *.class*) är däremot inte läsbara. Ett vanligt filtillägg för textfiler är *.txt*.

Studera följande två program. Båda läser in tal och skriver ut kvadratroten på dem. Lägg märke till likheter och skillnader.

```
1  import java.util.Scanner;
2
3  public class SquareRoot {
4      public static void main(String[] args) {
5          Scanner scanner = new Scanner(System.in);
6          while (scanner.hasNext()) {
7              double d = scanner.nextDouble();
8              System.out.println(Math.sqrt(d));
9          }
10     }
11 }
```

```
1  import java.io.File;
2  import java.io.FileNotFoundException;
3  import java.io.PrintWriter;
4  import java.util.Scanner;
5
6  public class SquareRootPersistent {
7
8      public static void main(String[] args) {
9          Scanner scanner = null;
10         try {
11             scanner = new Scanner(new File("numbers.txt"));
12         } catch (FileNotFoundException e) {
13             System.out.println("Couldn't find file numbers.txt");
14             System.exit(1);
15         }
16
17         PrintWriter output = null;
18         try {
19             output = new PrintWriter(new File("result.txt"));
20         } catch (FileNotFoundException e) {
21             System.out.println("Couldn't open file result.txt");
```

```
22              System.exit(1);
23          }
24
25          while (scanner.hasNextDouble()) {
26              double d = scanner.nextDouble();
27              output.println(Math.sqrt(d));
28          }
29          output.close();
30      }
31  }
```

Läsa från en textfil

I bägge programmen ovan använder man ett `Scanner-objekt` för att läsa tal. Det som skiljer är vad som skickas med som argument när man skapar `Scanner`-objektet. Här är exempel som visar hur man kan skapa en skanner som läser data från

- konsolfönstret:
  `new Scanner(System.in);`
- en fil med namnet *numbers.txt*:
  `new Scanner(new File("numbers.txt"));`
- en sträng:
  `new Scanner("2  0,67 25 100 3,42");`

På rad 11 i programmet ovan skapas ett `Scanner`-objekt som är kopplat till en fil med namnet *numbers.txt*. När väl `Scanner`-objektet är skapat är det bara att läsa som vanligt.

Tyvärr behövs det ytterligare kod omkring satsen där `Scanner`-objektet skapas (se rad 10-15). Vissa fel som kan inträffa, t.ex. att försöka öppna en fil som inte finns, måste man ta hand om det i en try catch-sats. Vi försöker öppna filen (rad 11) och om det inte går avbryts exekveringen av satserna i try-blocket och satserna i catch-blocket kommer att utföras (rad 13-14). Om filen inte kan öppnas har vi valt att skriva ut felutskrift och att avbryta exekveringen av programmet.

Detta med try catch-satser är ingenting som egentligen ingår i kursen. I Java snabbreferens finns mönster för hur du skapar `Scanner`-objekt på detta sätt.

Skriva på en textfil

För att skriva ut på en text fil kan man skapa ett `PrintWriter`-objekt och koppla det till en fil. På rad 19 i programmet ovan skapas `PrintWriter`-oblektet `output` som skriver ut på en fil med namnet *result.txt*. Om det inte redan finns någon sådan fil kommer den att skapas. Sedan är det bara att skriva ut som ”vanligt” men byt ut `System.out` mot `output`. Även i detta fall behövs en try-catch-sats omkring öppnandet av filen.

Anropet av `close` på rad 29 är viktigt. Utdata skrivs först ut på på en buffer i datorn och ut på filen när bufferten är full. Anropet av `close` medför att allt i bufferten skrivs ut på filen och att filen stängs på ett kontrollerat sätt.

Programmet ovan är inte så flexibelt. Filnamnen är hårdkodade i programkoden. Ett alternativ är att låta användaren välja filnamn.

Datorarbete

1. Provkör gärna programmet `SquareRootPersistent` som finns i paketet *squareroot*. (I projektet *Lab08* är filerna uppdelade i olika paket istället för att ligga i paketet *default package*.)
   Om du inte ser resultatsfilen så markera projektet och välj Refresh i File-menyn.

2. I resten av uppgifterna ska du läsa in koordinaterna för polygoners hörnpunkter och sedan rita dem i ett fönster. Börja med att skriva klassen `Polygon`. Lägg klassen i paketet *polygon*. Där finns redan en färdig klass `Point` som ska användas för polygonens hörnpunkter. `Polygon` har följande specifikation:

```
/** Skapar en kvadrat som kan ha upp till 10 hörnpunkter. */
public Polygon();

/** Lägger till en ny hörnpunkt med koordinaterna x, y. */
void addVertex(int x, int y);

/** Ritar polygonen i fönstret w. */
public void draw(SimpleWindow w);
```

   Ledning och anvsiningar:

   - Använd en vektor av typen `Point[]` för att lagra hörnpunkterna. Vektorn är från början tom och fylls på efterhand (vid anrop av `addVertex`). Därför behövs ytterligare ett attribut av typen `int` som håller reda på antal inlagda punkter.

3. I paketet *polygon* finns ett färdigt huvudprogram `OnePolygon` som läser in koordinater från konsolfönstret och ritar motsvarande polygon i ett ritfönster. Använd programmet för att testa din klass `Polygon`. Koden i main-metoden är bortkommenterad för att undvika kompileringsfel innan klassen `Polygon` finns. Tag bort kommentartecknen genom att markera den aktuella koden och välja Toogle Comment i Source-menyn. För att simulera filslutstecken (EOF) skriv Ctrl-D (Ctrl-Z på Windows).

4. Ändra i `OnePolygon` så att koordinaterna istället läses från en textfil. Två stycken förändringar behövs:

   1. Användaren ska själv få välja fil. Lös detta genom att läsa in namnet få filen.
   2. Ändra koden för att skapa `Scanner`-objektet så att det läser från textfilen istället.

   I projektet *Lab08* finns filen *polygon1* att testa programmet med.

5. Skriv ett nytt huvudprogram (klass med main-metode) som läser flera rader med polygoner:

   1. Utgå från programmet codeOnePolygon. Kopiera filen *OnePolygon.java* (högerklicka på projektet, sedan Copy och Paste). Döp den nya klassen till *ManyPolygon.java*.
   2. Nu när filen består av mer komplicerad data är det enklast att så länge det finns indata läsa hela rader och lagra den inlästa raden i en variabel av typen `String`. Sedan kan man skapa ett nytt `Scanner`-objekt som läser från den raden. För att läsa hela rader används metoderna `hasNext` och `nextLine`.

Checklista

Exempel på vad du ska kunna efter laborationen:

- Deklarera, skapa och använda vektorer som innehåller objekt.
- Förstå att data kan lagras i textfiler (och på andra sätt) mellan exekveringarna.
- Skriva programkod för att läsa från och skriva på textfiler.

Att diskutera

Klassen `Polygon` kan inte hantera polygoner med fler än 10 hörnpunkter. Vad händer om man försöker lägga till en 11:e punkt? Hur kan man lösa detta?

Laboration 9 – matriser, Memory-spel

Mål: Du ska träna på att att använda matriser.

Förberedelseuppgifter

- Läs avsnittet Bakgrund.
- Lös uppgift 1 under Datorarbete.

Bakgrund

Med Memory avses här spelet att hitta matchande par på en spelplan med kort. Spelet består av ett jämnt antal kort där det finns exakt två kort med samma motiv på framsidan. Spelet går till så att alla korten blandas och placeras ut på spelplanen med baksidan uppåt. Spelaren vänder på två kort i taget. Om de båda korten visar olika bilder måste de vändas upp och ner igen. Om de visar samma bild får de ligga kvar med framsidan upp. Spelet fortsätter tills spelaren lyckats hitta alla par.

Datorarbete

1. För att beskriva en (tvåsidig) bild av ett memorykort används den färdiga klassen `MemoryCardImage`.

   `MemoryCardImage`

   ```
   /** Skapar en tvåsidig bild av ett memorykort.
       Bilden på framsidan finns i filen med namnet frontFileName och
       bilden på baksidan i filen med namnet backFileName. */
   MemoryCardImage(String frontFileName, String backFileName);

   /** Returnerar bilden på framsidan. */
   Image getFront();

   /** Returnerar bilden på baksidan. */
   Image getBack();
   ```

   Klassen `MemoryCardImage` finns i ditt projekt *Lab09*. Titta på koden, se vilka metoder som finns och ungefär hur de är skrivna. Kan du förstå koden på ett ungefär?

2. Klassen `MemoryBoard` beskriver ett bräde med 16 memorykort. Ett ”skelett” till klassen finns i projektet *Lab09*. Din uppgift är att implementera den enligt specifikationen nedan.

   `MemoryBoard`

   ```
   /** Skapar ett memorybräde med size * size kort.
       backFileName är filnamnet för filen med baksidesbilden.
       Vektorn frontFileNames innehåller filnamnen för frontbilderna. */
   MemoryBoard(int size, String backFileName, String[] frontFileNames);

   /** Tar reda på brädets storlek. */
   int getSize();

   /** Hämtar den tvåsidiga bilden av kortet på rad r, kolonn c.
   Raderna och kolonnerna numreras från 0 och uppåt. */
   MemoryCardImage getCard(int r, int c);

   /** Vänder kortet på rad r, kolonn c. */
   ```

```
void turnCard(int r, int c);

/** Returnerar true om kortet r, c har framsidan upp. */
boolean frontUp(int r, int c);

/** Returnerar true om det är samma kort på rad r1, kolonn c1
    som på rad r2, kolonn c2. */
boolean same(int r1, int c1, int r2, int c2);

/** Returnerar true om alla kort har framsidan upp. */
boolean hasWon();
```

Att placera ut korten

När brädet skapas ska 8 st `MemoryCardImage`-objekt skapas. Varje sådant objekt ska placeras ut på två slumpmässiga platser på brädet. Om det redan finns ett objekt på en vald plats måste du välja en ny plats. Du kan alltså behöva dra slumptal flera gånger för att placera ut ett kort. En sådan algoritm kan informellt formuleras så här:

Upprepa följande för vart och ett av filnamnen i `frontFileNames`:

- Skapa ett `MemoryCardImage`-objekt med detta namn.
- Placera **två** referenser till detta objekt på brädet.
  Varje referens placeras ut så här:
  - Gissa (dra slumptal) värden på rad och kolumn för kortet.
    Kalla dessa värden r och c.
  - Så länge platsen (r, c) är upptagen på brädet, dra nya slumptal för r och c.
  - Därefter vet vi att platsen (r, c) är ledig.
    Det går därför bra att stoppa in en ny referens där.

Ledning

- Du måste alltså skilja på den bild av spelet som den som kör programmet har och hur det ser ut inuti klassen `MemoryBoard`. Spelaren ser 16 kort framför sig, men i själva verket representeras korten istället av de 8 `MemoryCardImage`-objekten.
- För att hålla reda på om ett kort har framsidan upp eller ej ska du använda en `boolean`-matris med samma dimensioner som brädet.

  Klassen `MemoryWindow` anropar metoden `frontUp` i `MemoryBoard` för att avgöra om ett givet kort ska ritas som upp- eller nedvänt. Det huvudprogram, som du kommer att skriva senare i laborationen, kommer också att använda den metoden för att avgöra om ett kort kan vändas upp eller ej.

  Medan du testar utplaceringen av kort i din `MemoryBoard`-klass (algoritmen ovan) kan det vara praktiskt att låta alla kort ritas uppvända från början. Då kan vi se att utplaceringen av korten fungerar som den ska. För sådan testning ska alltså metoden `frontUp` returnera `true` för alla platser på brädet.
- Vi vill gärna undvika alltför långa och stökiga metoder, och detta gäller särskilt konstruktorer. I kodskelettet finns därför en privat hjälpmetod `createCards`. Privata metoder syns inte i specifikationen, men genom att implementera metoden och använda den från konstruktorn blir konstruktorn kortare och koden mer lättläst.

  Det är inte nödvändigt att använda just denna privata metod: välj gärna en egen.

- När man utvecklar program implementerar man oftast inte en hel klass i taget, som det beskrivs här, utan det normala är att man vill testa mycket och ofta. De första testerna gör man helst innan man har hunnit skriva allt för mycket kod.

  För denna uppgift är det rekommenderat att implementera konstruktorn och metoderna `getSize` samt `getCard` och därefter gå vidare till `main`-metoden i `MemoryGame` (se uppgifterna 3 och 4) och där kontrollera att man kan skapa och rita upp ett bräde med alla kortens baksidor upp. När det fungerar implementerar man resten av `MemoryBoard` och för vaje metod som implementeras kan man göra ett enkelt test genom att anropa respektive metod från main-metoden.

3. För att visa korten och låta användaren klicka på ett kort finns den färdigskrivna klassen `MemoryWindow` (se specifikationen nedan). Klassen är en subklass till `SimpleWindow` vilket betyder att den har de attribut och metoder som finns i `MemoryWindow` plus allt som finns i `SimpleWindow`.

```
/** Skapar ett fönster som kan visa memorybrädet board. */
MemoryWindow(MemoryBoard board);

/** Ritar brädet med alla korten. */
void drawBoard();

/** Ritar kortet på rad r, kolonn c.
    Raderna och kolonnerna numreras från 0 och uppåt. */
void drawCard(int r, int c);

/** Tar reda på raden för senaste musklick. */
int getMouseRow();

/** Tar reda på kolonnen för senaste musklick. */
int getMouseCol();
```

   Vilka metoder som finns i `SimpleWindow` undersöker du enklast i ditt workspace. (I Eclipse-projektet *cs_pt* under katalogen *doc* kan man hitta filen `index.html`. Dubbelklicka på den för att få upp dokumentationen över de färdiga klasserna. Välj `SimpleWindow` i listan nere till vänster.)

   De metoder i `SimpleWindow` som du troligen behöver för att klara av uppgiften är följande:

```
/** Väntar ms millisekunder. */
static void delay(int ms);

/** Väntar tills användaren klickat på ett kort på brädet. */
void waitForMouseClick();
```

   Öppna `MemoryWindow` och läs igenom koden innan du går vidare.

4. Öppna klassen `MemoryGame` och skriv klart `main`-metoden som ska låta en person spela Memory. Korten ska visas i ett fönster och spelaren ska upprepade gånger välja två kort genom att klicka på dem. När alla kort visar framsidan är spelet slut och antal försök som krävdes ska skrivas ut.

```
public class MemoryGame {
    public static void main(String[] args) {
```

```
            String[] frontFileNames = { "can.jpg", "flopsy_mopsy_cottontail.jpg",
                "friends.jpg", "mother_ladybird.jpg", "mr_mcgregor.jpg",
                "mrs_rabbit.jpg", "mrs_tittlemouse.jpg", "radishes.jpg" };

            // Fyll i egen kod här
        }
    }
```

Vektorn `frontFileNames` innehåller de 8 filnamnen för framsidesbilderna. Bilden för kortens baksida finns i en fil med namnet *back.jpg*.

Ledning:

- Metoden `drawCard` i `MemoryWindow` ritar baksidan av det angivna kortet om det inte är vänt, och framsidan av kortet om det är noterat som vänt. Att vända ett kort är alltså i själva verket att ändra informationen om huruvida kortet är vänt eller ej, och därefter rita om kortet på nytt.
- Tänk på att spelaren måste hinna se på korten en stund innan de eventuellt vänds tillbaka.
- Om man klickar på ett redan uppvänt kort ska klicket ignoreras.

Frivilliga extrauppgifter

Extrauppgifterna har ingen inbördes ordning och beror inte av varandra.

5. Istället för att skriva ut antalet försök som krävdes till terminalfönstret kan det vara idé att visa resultatet i en ruta istället. Experimentera med Javas inbyggda klass `JOptionPane` för att göra detta. Testa t.ex:

   ```
   JOptionPane.showMessageDialog(null, "meddelande", "titel",
       JOptionPane.INFORMATION_MESSAGE);
   ```

6. Modifiera ditt spel så att användaren kan välja att köra en runda till, utan att stänga av mellan gångerna.

7. Inför ett rekord (”highscore”) som visas efter varje spelomgång och skrivs till fil.

Laboration 10 – använda klassen ArrayList, spelkort

Mål: Du ska träna på att använda klassen `ArrayList`. Du ska också prova att formulera en algoritm steg för steg.

Förberedelseuppgifter

- Läs igenom texten i avsnittet Bakgrund.

Bakgrund

I den här laborationen ska du skriva klasser som kan användas i kortspel. Ett spelkort representeras av den färdigskrivna klassen `Card` som har följande specifikation. Observera att klasserna på den här laborationen skiljer sig från exemplen i läroboken (Downey & Mayfield: Think Java).

Card

```
/** konstanter för färgerna */
static final int CLUBS = 0;
static final int DIAMONDS = 1;
static final int HEARTS = 2;
static final int SPADES = 3;

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

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

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

/** Returnerar en läsbar representation av kortet, till exempel
    "spader ess". */
String toString();
```

Både färg och valör representeras av heltal. Konstanterna är till för att man inte ska behöva komma ihåg att 3 betyder spader. Utanför klassen `Card` skriver man `Card.SPADES` för spader.

Klassen som beskriver en kortlek ska du själv implementera under laborationen:

Deck

```
/** Skapar en blandad kortlek med 52 kort. */
Deck();

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

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

Kortleken är en ”engångskortlek” och man måste skapa ett nytt `Deck-objekt` om man vill ha en nyblandad kortek.

När man implementerar en klass måste man skilja på hur den ska användas och hur den är implementerad inuti. Kommentarerna till de publika metoderna beskriver *vad* en metod gör, inte *hur*. Första tanken du får är kanske att i konstruktorn skapa en lista med 52 `Card`-objekt och sedan blanda korten genom att möblera om slumpmässigt i listan. I metoden `getCard` behöver

man då bara ta bort och returnera första (eller sista) kortet från lista. Men en enklare lösning är att låta listan förbli oblandad, men att i `getCard` ta bort ett kort på en slumpmässigt vald plats.

Datorarbete

1. I paketet *cards* finns en påbörjad klass `Deck`. Skriv färdigt klassen. Använd en `ArrayList` för att hålla reda på de 52 korten.

   För att testa ditt program ska du använda det färdiga programmet `TestCardDeck` som finns i paketet `test` (titta gärna i denna klass för att se vad den gör.).

2. I den här deluppgiften ska du använda klassen `Deck` för att lägga en patiensen 1–2–3. Den läggs på följande sätt: Man tar en blandad kortlek och lägger 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.[2] Men det är betydligt enklare att uppskatta sannolikheten genom att lägga patiensen många gånger och räkna antalet gånger som den går ut.

   Skapa ett huvudprogram (en klass med `main`-metod). I main-metoden kommer du att skriva ett kod som simulerar patiensen ett stort antal gånger och uppskattar sannolikheten för att den går ut. Utveckla programmet stegvis genom att följa dessa steg. *Kör programmet och se till att det fungerar efter varje steg.*

   1. Skapa kortleken. Dra alla korten från en kortlek och skriv ut korten.
   2. Istället för att skriva ut korten ska du visa dem i ett `CardWindow`-fönster med plats för ett kort. Klassen `CardWindow` är en specialisering av `SimpleWindow` och är ett fönster som kan visa spelkort. Specifikationen finns sist i laborationen. Lägg in en fördröjning, t.ex. `CardWindow.delay(2000)` så att du hinner se de olika korten.
   3. Lägg till en räknare som får värdena 1-2-3-1-2-3..... Skriv ut räknarens värde i fönstret. För att omvandla ett heltal till en sträng kan du använda den statiska metoden `valueOf(int x)` i klassen `String`.
   4. Avbryt loopen om patiensen inte gick ut.
   5. Lägg patiensen ett stort antal gånger och räkna hur många gånger den inte gick ut. Skriv ut detta antal.

      Tips!

      - Du kommer nog att behöva experimentera lite med hur många ”ett stort antal gånger” är. Det är därför praktiskt att införa en konstant som anger antalet upprepningar. Låt den exempelvis heta `NBR_ITERATIONS`, inledningsvis ha värdet 10000 och deklareras som konstant i din klass:

        ```
        private static final int NBR_ITERATIONS = 10000;
        ```

      - Kommentera bort de rader som har med `CardWindow` att göra för att göra exekveringen snabbare.
   6. Beräkna sannolikheten för att patiensen går ut och ändra utskriften så att denna sannolikhet skrivs ut istället för antal misslyckade försök. Sannolikheten bli ungefär 0,008165.

[2] Se ”Fråga Lund om matematik”, `www.maths.lth.se/query/answers`, sök efter ”patiens” på sidan.

3. I kortspel måste man hålla reda på de kort man har på handen. Implementera därför följande klass. Använd en `ArrayList` för hålla reda på korten.

   `Hand`

   ```
   /** Skapar ett objekt som kan hålla reda på en spelares kort på hand.
       Från början är handen tom. */
   Hand();

   /** Sätter in kortet c. Korten sorteras in efter stigande valör. */
   void insert(Card c);

   /** Tar bort och returnerar kortet på posititon pos från handen.
       Korten numreras från 0 och uppåt. */
   Card remove(int pos);

   /** Returnerar antalet kort. */
   int nbrCards();

   /** Beräknar summan av kortens valörer. Om ace14 har värdet true ska
       ess räknas som 14, annars som 1. */
   int rankSum(boolean ace14);

   /** Visar korten i fönstret w. */
    void display(CardWindow w);
   ```

   Klassen `CardWindow` är en specialisering av `SimpleWindow` och är ett fönster som kan visa spelkort. Specifikationen finns sist i laborationen.

4. Skriv en klass med en `main`-metod som använder metoderna i klassen `Hand`. Programmet ska genomföra en spelomgång av en mycket förenklad variant av spelet 21. Tre kort dras från en blandad kortlek. Därefter ska summan av de tre kortens valörer beräknas. Ess ska räknas som 1. Om summan är mindre eller lika med 21 har man vunnit, i annat fall förlorat.

   Programmet ska visa de tre korten i ett lagom stort fönster. I detta fönster ska också summan samt uppgift om man vunnit eller ej skivas ut.

   Kontrollera att korten skrivs ut i ordning (efter växande valör) när du kör programmet.

Checklista

Exempel på vad du ska kunna efter laborationen:

- Deklarera, skapa och använda listor av typen `ArrayList`.
- Förstå att hur en klass används (vad dess metoder gör) är en sak och hur klassen är implementerad är en annan sak.
- Formulera algoritmer i Java.
- Utveckla algoritmer och program genom att börja med enklast möjliga lösning och sedan utveckla stegvis.

Frivilliga extrauppgifter

- Utveckla spelet 21 eller skriv program fört något helt annat kortspel (eller någon patiens). Hitta på någon egen metod i klassen `Hand` som skulle kunna vara användbar i något annat spel. Lägg till ny metoder i klassen `Hand` ifall det behövs.

Klassen `CardWindow`

`CardWindow`

```
/** Constructs a window with rows rows and cols columns which can
    display playing cards. The window has the title title. */
 CardWindow(int rows, int cols, String title);

/** Displays the cards in the list cardList. */
void displayCards(List<Card> cardList);

/** Displays the cards in the array cardArray. */
void displayCards(Card[] cardArray) ;

/** Displays the card aCard in line r, column c. Rows and columns are
    numbered from 0 and up. */
void displayCard(Card aCard, int r, int c) ;

/** Waits for mouseclick. Returns the card at the mouse click. If no
    card exist at the mouse click null is returned.  */
Card getClickedCard();

/** Waits for mouseclick. Deletes and returns the card at the mouse
     click. If no card exist on the mouse click null is returned. */
Card deleteCard() ;

/** Wait for a specified time.  */
static void delay(int ms);

/** Displays the string s in the window. */
void displayText(String s);
```

Laboration 11 – arv, TurtleRace

Mål: Du ska förstå grundläggande användning av arv. Du ska också få mer träning på att använda vektorer och `ArrayList`.

Förberedelseuppgifter

- Läs avsnittet Bakgrund.
- Studera klassen `RaceWindow`. Om något är oklart, fråga din handledare vid laborationens början.

Bakgrund

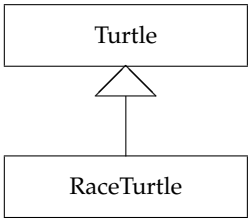
Under den här laborationen ska du skriva ett program där du låter sköldpaddor tävla mot varandra i en kapplöpning.

Vi använder klassen `SimpleWindow` på ett annorlunda sätt än tidigare: vi har här specialiserat den genom subklassen `RaceWindow`. Ett `RaceWindow` fungerar som ett vanligt `SimpleWindow` (eftersom det ärver från `SimpleWindow`) men med tillägget att en kapplöpningsbana ritas upp då fönstret skapas.

Uppgiften går ut på att skriva ett program som simulerar en (slumpmässig) kapplöpning mellan sköldpaddor (`Turtle`-objekt). En sköldpadda tar sig fram genom simulerade tärningskast (dvs. slumpmässiga steg framåt mellan 1 och 6), med var sitt ”kast” i varje runda.

En klass `RaceTurtle` ska representera en sköldpadda som kan simulera en kapplöpning. `RaceTurtle` ska ärva från klassen `Turtle` och därmed kan en `RaceTurtle` göra allt som en vanlig `Turtle` kan. Därtill ska en `RaceTurtle` innehålla en slumptalsgenerator och ett startnummer samt metoderna `raceStep()` och `toString()`:

```
/**
 * Skapar en sköldpadda som ska springa i fönstret w och som har start-
 * nummer nbr. Sköldpaddan startar med pennan nere och vänd åt höger.
 */
RaceTurtle(RaceWindow w, int nbr);

/**
 * Låter sköldpaddan gå framåt ett steg. Stegets längd ges av ett
 * slumptal (heltal) mellan 1 och 6.
 */
void raceStep();

/**
 * Returnerar en läsbar representation av denna RaceTurtle,
 * på formen "Nummer x" där x är sköldpaddans startnummer.
 */
String toString();
```



Datorarbete

1. Inkludera din `Turtle` från laboration 6 genom att:
   1. Högerklicka på projektet *Lab11*, välj *Build Path* → *Configure Build Path*.
   2. Klicka på fliken *Projects*.
   3. Klicka på *Add...*, markera *Lab06* och klicka *OK*.
   4. Klicka på *OK* för att lämna dialogen.

2. Implementera klassen `RaceTurtle` enligt beskrivning och specifikation ovan. När du implementerar konstruktorn för `RaceTurtle` måste du beräkna dess position. Det finns två hjälpmetoder i `RaceWindow`, som beroende på startnummer returnerar en lämplig x- respektive y-position. På så sätt får alla sköldpaddor en egen bana.

3. I klassen `RaceTurtleTest` finns en main-metod som skapar en sköldpadda och låter denna genomföra ett eget lopp utan motståndare. Klassen kan användas som ett första test av din `RaceTurtle`. Avkommentera och studera koden – det är meningen att du ska förstå vad den gör. Kör sedan programmet för att se om sköldpaddan går i mål som förväntat.

4. Gör en ny klass `TurtleRace` för att genomföra ett lopp enligt följande (det är lämpligt att inspireras av testprogrammet):
   - Skapa åtta sköldpaddor och lagra dessa i en vektor. Låt dem ha sin kapplöpning i ett och samma `RaceWindow`.
   - Efter loppet ska första, andra och tredje plats skrivas ut (enligt exemplet nedan). Det är ett krav att utskriften använder sig av `toString`-metoden.

     ```
     På plats 1: Nummer 5
     På plats 2: Nummer 6
     På plats 3: Nummer 1
     ```

   - **Tips:** När en sköldpadda går i mål kan du sätta in den i en separat lista av typen `ArrayList`. När listan har samma storlek som vektorn har alla sköldpaddorna gått i mål, och då görs utskriften ovan. Tänk på att sköldpaddor som gått i mål inte ska fortsätta ta steg.
   - Små orättvisor i resultatberäkningen är acceptabla. Kanske upptäcker du att din lösning gynnar sköldpaddor med lägre startnummer över de med högre startnummer. Förklara för din handledare vari orättvisan består, och diskutera gärna hur man kan åtgärda den.
   - Det är därmed inte heller nödvändigt att hantera delade placeringar.

5. Provkör ditt sköldpaddslopp. Använd en fördröjning, t. ex. `RaceWindow.delay(10)`, så man hinner se hur loppet fortskrider. Kontrollera att rätt placeringar skrivs ut.

Checklista

I den här laborationen har du övat på att

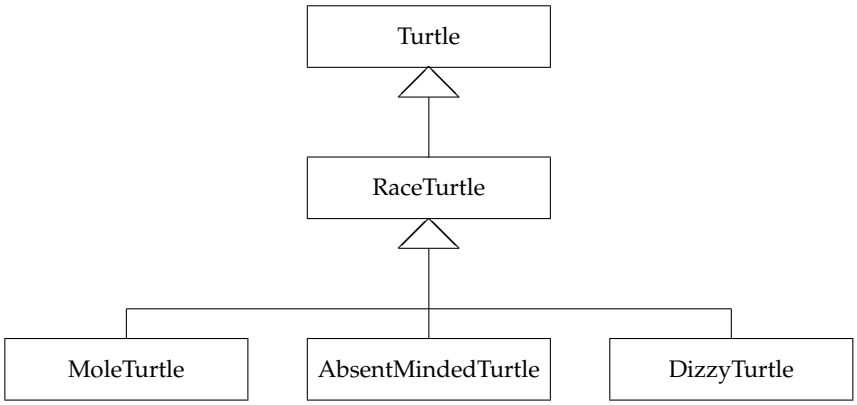
- skriva subklasser som ärver egenskaper från en superklass.
- använda vektorer och `ArrayList` för att lagra objekt.

Frivillig extrauppgift

Uppgiften går ut på att skriva en eller flera klasser som beskriver olika slags tävlande sköldpaddor, samt att modifiera huvudprogrammet för att genomföra en sådan kapplöpning. Klassen `RaceTurtle` ska representera det som samtliga tävlingssköldpaddor har gemensamt. Nu ska emellertid metoden `raceStep` överskuggas (override) av subklassernas mer specifika definition av metoden. Vi behöver en subklass för varje slags sköldpadda, och bilden nedan visar arvshierarkin.

Här är exempel på olika slags tävlande sköldpaddor (subklasser till `RaceTurtle`):

- `MoleTurtle`, som beskriver en ”mullvadssköldpadda”, d.v.s. en tävlande sköldpadda som då och då går under jorden (genom att lyfta pennan slumpmässigt).
- `AbsentMindedTurtle`, som är en tankspridd sköldpadda. Graden av tankspriddhet anges i procent när sköldpaddan skapas, och sannolikheten att en tankspridd sköldpadda skall gå framåt bestäms av tankspriddhetsgraden. Exempel: en tankspriddhetsgrad på 34 procent ska medföra att sköldpaddan i 34 procent av fallen glömmer att ta sitt steg.
- `DizzyTurtle`, som beskriver en yr tävlingssköldpadda, som vinglar när den skall ta sig framåt. När dessa sköldpaddor skapas ska graden av yrsel (från 1 till 5) anges, och deras förmåga att hålla kursen skall bestämmas av yrselgraden. Du behöver inte göra någon avancerad simulering av yrseln, men en sköldpadda med högre yrselgrad bör vingla mer än en mindre yr sköldpadda.



1. Implementera en eller flera av tävlingssköldpaddorna som beskrivs ovan. (Eller hitta på egna varianter). Skapa en ny klass för varje typ av sköldpadda och låt varje klass ärva från `RaceTurtle`. Varje sådan subklass ska ha sin egen definition av metoderna `raceStep()` och `toString()` – dessa ska alltså implementeras på nytt.

   Observera att `raceStep()` och `toString()` inte är abstrakta i `RaceTurtle`. De har alltså redan en implementation. Det gör att du själv måste komma ihåg att överskugga dessa i varje subklass, men de går samtidigt att utnyttja för att slippa duplicera kod. Exempelvis kan kan man från subklassernas `raceStep`-metoder anropa `super.raceStep()` för att genomföra själva steget, förutom den kod som är specifik för varje subklass (lyfta/sänka pennan, jämföra tankspriddhet, välja riktning).

2. Modifiera klassen `TurtleRace`: genom att
   - Skapa åtta sköldpaddor av slumpmässig typ.
   - Då `AbsentMindedTurtle` ska skapas ska tankspriddhetsgraden slumpas fram mellan 0 och 100.
   - Då `DizzyTurtle` ska skapas ska yrselgraden slumpas fram mellan 1 och 5.

- Innan loppet börjar ska hela startuppställningen skrivas ut enligt exemplet nedan, ett krav är att detta sker genom respektive sköldpaddas `toString`-metod. I exemplet är sköldpadda 5 och 8 ”vanliga” sköldpaddor av typen `RaceTurtle`.

```
Nummer 1 - MoleTurtle
Nummer 2 - AbsentMindedTurtle (45% Frånvarande)
Nummer 3 - AbsentMindedTurtle (43% Frånvarande)
Nummer 4 - DizzyTurtle (Yrsel: 3)
Nummer 5
Nummer 6 MoleTurtle
Nummer 7 AbsentMindedTurtle (57% Frånvarande)
Nummer 8
```

- Efter loppet ska första, andra och tredje plats skrivas ut (enligt exemplet nedan).

```
På plats 1: Nummer 5 - MoleTurtle
På plats 2: Nummer 6 - MoleTurtle
På plats 3: Nummer 4 - DizzyTurtle (Yrsel: 3)
```